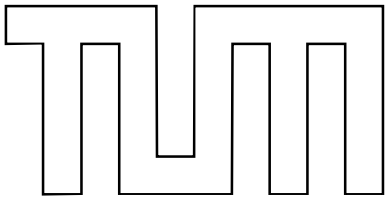
INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN



Fortgeschrittenenpraktikum

Objektorientierte Modellierung von Workstations für ein integriertes Systemmanagement

Holger Sirtl

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering
Betreuer: Alexander Keller

Inhaltsverzeichnis

1 Motivation 4

2 Common Object Request Broker Architecture (CORBA) 5
2.1 Object Management Group 5
2.2 Object Management Architecture 5
2.2.1 Object Request Broker 5
2.2.2 CORBAservices 6
2.2.3 CORBAfacilities 6
2.2.4 Application Objects 6
2.3 Common Object Request Broker Architecture (CORBA) 6
2.3.1 Die Dienste des ORB 6
2.3.2 Der Aufbau des Object Request Broker in CORBA 7
2.3.3 Objekte 9
2.4 CORBAservices 9
2.5 CORBAfacilities 10
2.6 Application Objects 11
2.7 SOM/DSOM 11
2.7.1 SOM als „Objektmanager“ 11
2.7.2 Das SOM-Klassenkonzept 11
2.7.3 Distributed SOM (DSOM) 12
2.8 Generelle Entwicklung und Bereitstellung eines CORBA-Servers 13

3 Gewinnung eines geeigneten Objektmodells 14
3.1 Konkretes Vorgehen unter Verwendung der OMT 14
3.1.1 Die Object Modeling Technique (OMT) 14
3.1.2 Werkzeuge zur Unterstützung der Modellierung 14
3.2 Analyse 14

4 Design und Implementierung 16
4.1 Die Rolle der Verschiedenen Designmodelle 16
4.1.1 Das Objektmodell 16
4.1.2 Das Dynamikmodell 16
4.1.3 Das Funktionsmodell 16
4.2 „Naive“ Übersetzung der SNMP-MIB in ein Objektmodell 16
4.3 Bewertung der ersten Objekt-MIB 21
4.4 Optimierung des Modells 22
4.4.1 Neue Superklassen für gemeinsame Attribute und Operationen 22
4.4.2 Neue Unterklassen anstelle von Typvariablen 23
4.4.3 Operationen anstelle von „Pushbutton“-Variablen 23
4.4.4 Möglichst realitätsgetreue Modellierung der Objektbeziehungen 23
4.4.5 Neue Variablentypen für Variablen mit eingeschränktem Wertebereich 24
4.5 Implementierung 26
4.5.1 Implementierungsbedingte Nachbesserungen des Objektmodells 26
4.5.2 Nachbearbeitung der von StP gelieferten IDL-Schnittstellenbeschreibungen 28

5 Arbeiten mit dem StP/OMT-Softwarepaket 32
5.1 Die Funktionalität und Aufbau des StP/OMT-Softwarepaketes 32
5.2 Der StP Desktop 33
5.3 Die Editoren 34
5.4 Die Informationsspeicherung in StP 35
5.4.1 Die Systemdateien 35
5.4.2 Das Repository 36
5.5 Der Entwurf eines OMT-Modells 36
5.6 Beurteilung 37

6 Ausblick 38

Anhang A: Abkürzungen 39

Anhang B: Teil der OMT-Notation nach Rumbaugh 40

Anhang C: Literatur 41

Anhang D: Abbildungsverzeichnis 42

Anhang E: Die Objektmodelle 43

1 Motivation

Die Bedeutung integrierten Netz- und Systemmanagements hat mit der rasanten Entwicklung und Verbreitung von Kommunikationssystemen, Netzverbunden und verteilten Systemen in den vergangenen Jahren stark zugenommen. Dabei hat sich das Internet-Management auf der Basis des *Simple Network Management Protocol* zu einem Quasi-Standard entwickelt. Es bildet die Basis für die meisten heute verwendeten Managementlösungen. Inzwischen erweist sich das Internet-Management als in einigen Bereichen zu inflexibel. Organisations- und Funktionsmodell sind kaum bis überhaupt nicht implementiert. Das Informationsmodell ist einfach und nur relativ unstrukturiert gehalten.

Mit der *Common Object Request Broker Architecture* (CORBA) der *Object Management Group* (OMG) steht nun ein offener Ansatz zur Entwicklung verteilter, objektorientierter Systeme in heterogenen Netzen zur Verfügung. CORBA kann damit als Basisarchitektur für Systemmanagement verwendet werden. Schwerpunkt dieser Arbeit war die Erstellung eines für die Belange des Systemmanagements geeigneten Objektmodells und die anschließende Implementierung einiger ausgewählter Teile des Modells.

Abbildung 1 zeigt schematisch die konkrete Vorgehensweise. Als Basis für die Modellerstellung diente die bereits bestehende SNMP-MIB. Diese wurde zunächst „mechanisch“ in eine erste Objekt-MIB übertragen. Als Modellierungswerkzeug wurde dabei das *Software through Pictures* (StP) CASE-Tool verwendet. Die erste Objekt-MIB wurde dann mit dem Ziel optimiert, objektorientierte Prinzipien (z.B. möglichst gute Abbildung der Realität unter Verwendung von Vererbungs- und Enthaltenseinsbeziehungen) und die Belange des Systemmanagements gleichermaßen zu erfüllen. Als Ausgabe lieferte StP CORBA-IDL-Schnittstellenbeschreibungen. Diese und entsprechende Teile des bestehenden SNMP-Agentencodes waren die Eingaben für die Implementierung. Als Werkzeug hierfür diente das *SOMobjects Developer Toolkit* von IBM.

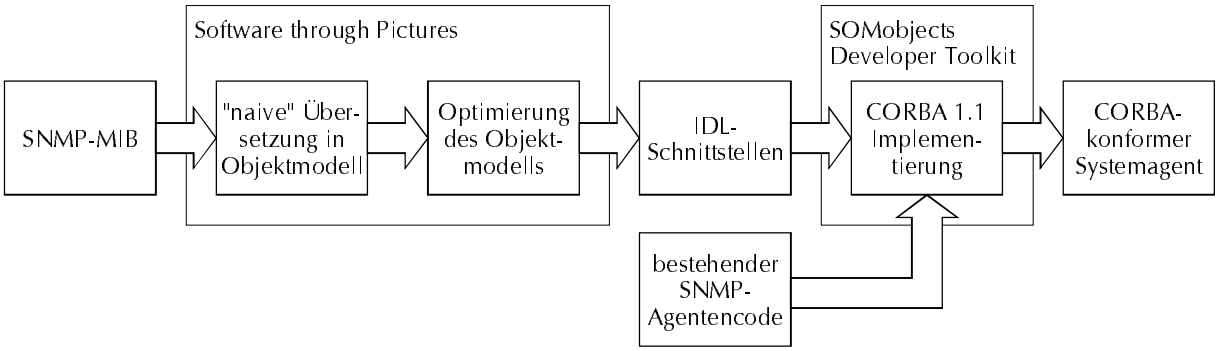


Abbildung 1: Vorgehensweise zur Gewinnung eines CORBA-konformen Systemagenten aus der ASN.1-Schnittstellenbeschreibung und dem bestehenden SNMP-Agentencode

Kapitel 2 gibt einen Überblick über die der Modellierung zugrundeliegende Systemarchitektur: die *Common Object Request Broker Architecture.*

Kapitel 3 und 4 beschreiben dann, wie aus der bestehenden SNMP-MIB ein Objektmodell gewonnen wurde und auf diesem dann einige Optimierungsschritte durchgeführt wurden. Am Ende von *Kapitel 4* stehen dann noch einige Erläuterungen zu Implementierung von Teilen des Systemagenten.

Das verwendete Softwarepaket „Software through Pictures“ steht im Mittelpunkt von *Kapitel 5.* Es wird kurz beschrieben, wie mit diesem CASE-Tool die Objektmodellierung abläuft.

Kapitel 6 rundet die Arbeit dann mit einem kurzen Ausblick auf die Möglichkeiten und die Erweiterbarkeit eines CORBA-basierten Systemmanagements ab.

2 Common Object Request Broker Architecture (CORBA)

2.1 Object Management Group

Das Ziel der 1990 gegründeten Object Management Group (OMG), eines herstellerübergreifenden Konsortiums, war von Anfang an die Bereitstellung einer geeigneten Architektur für die Verteilung und Zusammenarbeit objektorientierter Softwarebausteine in heterogenen und vernetzten Systemen. Als Ergebnis dieser Bemühungen entstand die Object Management Architektur.

2.2 Object Management Architecture

Die Idee, die hinter *distributed object computing* steckt, ist die folgende: Objekte beliebigen Typs sollen zu beliebigen Zeitpunkten Dienste anderer Objekte in Anspruch nehmen können, ohne darauf achten zu müssen, wo sich beide Objekte befinden und wie sie ihre jeweilige Aufgabe erfüllen (d.h. die Art und Weise, wie die Objekte implementiert sind, soll keine Rolle spielen).

Die Object Management Group beschreibt in ihrem *Object Management Architecture Guide* eine Architektur zur Realisierung verteilter Anwendungen. Die Object Management Architecture gliedert sich in vier Teile:

- Object Request Broker
- CORBAservices
- CORBAfacilities
- Application Objects

Abbildung 2 zeigt eine schematische Darstellung der *Object Management Architecture*:

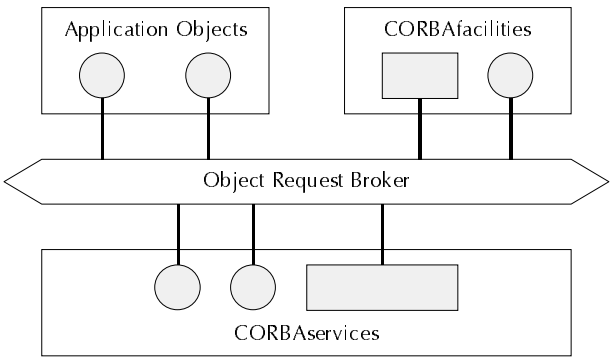


Abbildung 2: Die OMG Object Management Architecture

2.2.1 Object Request Broker

Wie schon in *Abbildung 2* ersichtlich, spielt der Object Request Broker (ORB) die zentrale Rolle in der OMA. Im Objektmodell der OMG bilden ein sogenannter Request und sein zugehöriges Resultat das zentrale Kommunikationsinstrument zwischen Objekten. Der Object Request Broker stellt Mechanismen zur Verfügung, die eine Objektinteraktion dergestalt vornehmen, daß Objektaufrufe transparent für den Aufrufer durch das Netz geschleust werden. Für den Aufrufer ist es nicht erkennbar, welche Maschine im Netz seine Anfrage bearbeitet, geschweige denn muß er auf spezielle Eigenarten (z.B. Implementierung, Programmiersprache) Rücksicht nehmen.

2.2.2 CORBAservices

CORBAservices bieten Basisfunktionen zur Verwaltung von Objekten im Netzwerk an. Insbesondere fallen hierunter alle Funktionen zur Erzeugung und Verwaltung von Klassen und Instanzen sowie die Möglichkeit, bei Bedarf persistente Objekte zu erzeugen. Im Vergleich zu den CORBAfacilities sind die CORBAservices also eher systemorientiert.

CORBAservices stellen Dienste bereit, die, gäbe es keine CORBAservices, wohl von jeder Anwendung, die über den ORB kommunizieren will, getrennt implementiert werden müßten. Die OMG hat dabei festgelegt, welche Services in einem CORBA-konformen ORB vorhanden sein müssen. Genauere Informationen zu CORBAservices finden sich in *Kapitel 2.4*.

2.2.3 CORBAfacilities

Common Facilities bieten all die Funktionen, die generellen Charakter haben und von einer Vielzahl von unterschiedlichen Applikationen genutzt werden. In diesem Sinn haben sie den Charakter einer allgemein benutzbaren Klassenbibliothek, sind im Vergleich zu den CORBAservices eher endbenutzerorientiert.

Beispiele hierfür sind etwa Hilfsfunktionen, die von mehreren verschiedenen Anwendungen genutzt werden. *Kapitel 2.5* enthält genauere Informationen zu den CORBAfacilities.

2.2.4 Application Objects

Application Objects sind all die unzähligen Anwendungen, die basierend auf der OMA Dienste anbieten bzw. solche des beschriebenen Systems nutzen.

Zu den Anwendungen, die voraussichtlich am schnellsten die Reichhaltigkeit der hier beschriebenen Architektur nutzen werden, zählen CAD-Anwendungen, CASE-Systeme und Netzwerk-Management-Anwendungen.

2.3 Common Object Request Broker Architecture (CORBA)

2.3.1 Die Dienste des ORB

Ein ORB, der die CORBA-Spezifikationen erfüllt, sollte folgende Dienste zur Verfügung stellen:

- **Name Service**
  gestattet es, Objekte im Netzwerk eindeutig zu benennen, d.h. der Name Service lokalisiert über die Bezeichnung ein Zielobjekt im Netz.

- **Request Dispatch Service**
  stellt für jeden Request fest, welche Methode(n) in welchem Objekt aufzurufen ist/sind.

- **Parameter Encoding**
  transferiert Aufruf- bzw. Rückgabeparameter aus einer maschinenspezifischen Darstellung in eine Netzwerkdarstellung bzw. umgekehrt.

- **Delivery Services**
  gewährleistet eine korrekte Zustellung von Requests bzw. Ergebnissen unter Verwendung von Transportprotokollen wie TCP/IP. Des weiteren wird hier eine Synchronisation parallel gestellter Anfragen durchgeführt.

- **Synchronization**
  gewährleistet, daß ein Diensterbringer dem Anfrager in zeitlich und inhaltlich geeigneten Art und Weise antwortet (z.B. synchron / asynchron)

- **Activation**
  letztlich der Aufruf einer Methode; darüber hinaus Verwaltungsfunktion bei persistenten Objekten, bei denen vor Aufruf einer Methode zunächst der aktuelle Status ermittelt und nach dem Aufruf gesichert werden muß, z.B. bei Objekten deren Informationen außerhalb in „Fremdspeichern" gehalten werden (z.B. in Dateien oder Datenbanken)

- **Exception Handling**
  Funktionen zur Behandlung von „Ausnahmesituationen", z.B. Ressourcenneustart oder Ressourcensicherung, wenn bei Ausführung eines Objektes eine Ressource ausfällt.

- **Security Services**
  gewährleisten eine eindeutige gegenseitige Identifikation und Authentifizierung von Kommunikationspartnern.

2.3.2 Der Aufbau des Object Request Broker in CORBA

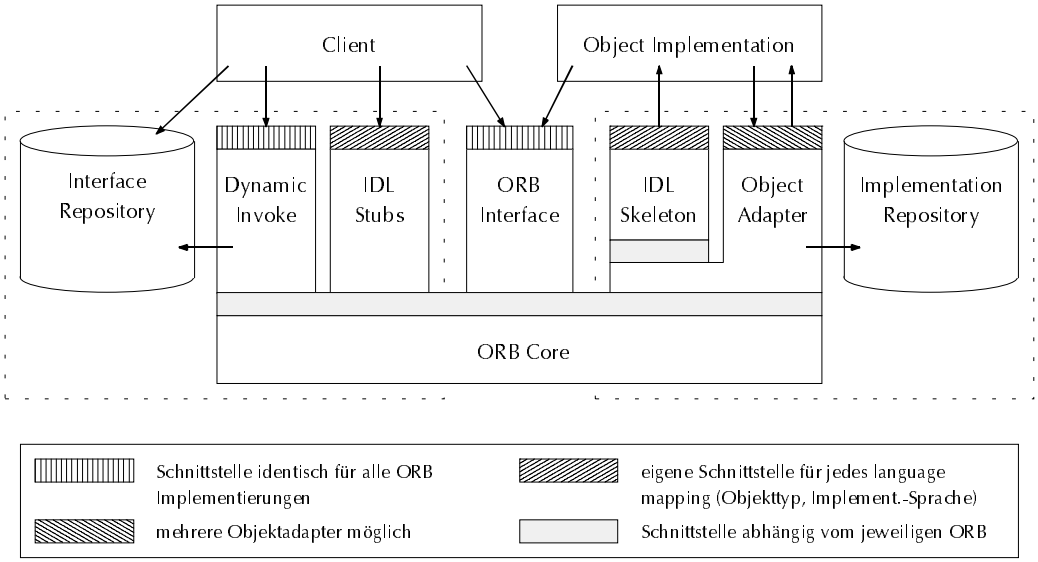


Abbildung 3: Die Architektur eines Object Request Brokers

Der ORB kann in folgende Bestandteile aufgegliedert werden:

- ORB Core
- IDL Stub Interface
- Dynamic Invocation Interface
- ORB Interface
- IDL Skeleton
- Object Adapter

2.3.2.1 ORB Core

Dies ist der Kern des ORB. Hier findet die Kommunikation statt, d.h. zum Beispiel die Weiterleitung der Requests zum Zielobjekt, die Übergabe der Parameter usw. Im Wesentlichen sind dies also die in Kapitel 2.3.1 (*Die Dienste des ORB*) beschriebenen Dienste.

2.3.2.2 IDL Stub Interface

Bestimmte Applikationen erfordern, daß von jedem verwendeten Objekttyp eine Schnittstelle, genannt *IDL stub*, während der Übersetzungsphase in die Applikation eingebracht wird. Die *stubs* führen Aufrufe an den ORB aus, indem sie Schnittstellen verwenden, die privat (und ggf. optimiert) für den jeweils verwendeten *ORB core* sind.

2.3.2.3 Dynamic Invocation Interface

Die wesentliche Neuerung im ORB-Konzept stellt die dynamische Aufrufschnittstelle dar. Ein Client hat hiermit die Möglichkeit, aus einem Interface Repository zur Laufzeit die Schnittstellenbeschreibung eines Objektes zu erfragen. Basierend auf dem Ergebnis dieser Abfrage kann er einen Aufruf einer Methode dieses Objektes erzeugen. Eine statische Erzeugung von objektspezifischen Aufrufrümpfen ist nicht mehr erforderlich, nicht einmal das Wissen, wo sich ein Objekt befindet.

Das aufgerufene Objekt kann nicht mehr erkennen, auf welche Art (über IDL Stubs oder über das Dynamic Invoke Interface) der Request gemacht wurde.

2.3.2.4 ORB Interface

Das ORB Interface ist die Schnittstelle, die direkt auf den ORB aufsetzt und für alle ORBs gleich ist und unabhängig ist von der jeweiligen Objektschnittstelle oder dem jeweiligen Objektadapter. Da der größte Teil der ORB-Funktionalität von Objektadapter, Stubs, Skeleton oder dynamischen Aufrufen bereitgestellt wird, bleiben nur wenige Operationen, die für alle Objekte gleich sind. Diese Operationen sind gleichermaßen für Clients und Objektimplementierungen nützlich. Beispiele für solche Operationen sind etwa Operationen zu Typumwandlungen.

2.3.2.5 IDL Skeleton

IDL Skeletons (oder Implementation Skeletons) stellen die Verbindung her zwischen Objektadapter und den Objektimplementierungen. Für jedes *language mapping* und ggf. für verschiede Objektadapter existieren solche Schnittstellen zwischen Objektreferenz und Objektimplementierung. Über diese *IDL Skeletons* ruft der Objektadapter Methoden der Objektimplementierungen auf.

2.3.2.6 Object Adapter

Ein Objektadapter ist die wichtigste Möglichkeit für eine Objektimplementierung, Dienste des ORB in Anspruch zu nehmen. Ein Objektadapter exportiert eine öffentliche Schnittstelle zu der Objektimplementierung und eine private Schnittstelle zum *Skeleton*. Objektadapter sind auf einer privaten ORB-abhängigen Schnittstelle aufgebaut.

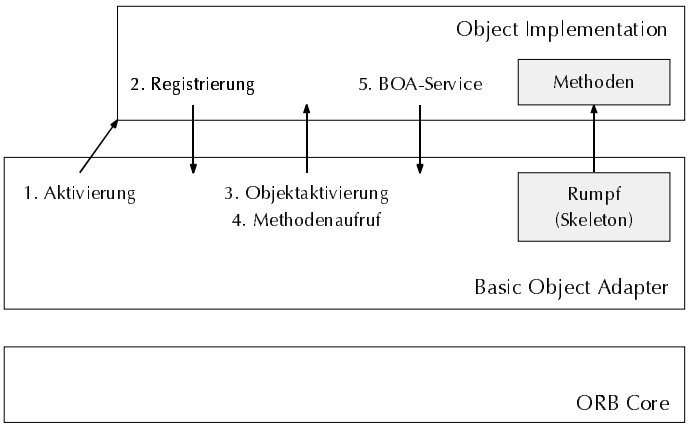


Abbildung 4: Funktionsprinzip des Basic Object Adapter

Objektadapter stellen folgende Funktionen zur Verfügung:

- Generierung und Interpretation von Objektreferenzen[1]
- Methodenaufruf
- Sicherstellung der Sicherheit von Interaktionen
- Aktivierung und Deaktivierung von Objekten und Implementierungen
- Zuordnung von Objektreferenzen zu entsprechenden Objektimplementierungen
- Registrierung der Implementierungen

2.3.3 Objekte

Die Implementierung von CORBA-Objekten erfolgt durch Prozesse, sogenannte Server. Dafür existieren mehrere Implementierungsvarianten:

- **Shared Server**
  Beim *shared-server*-Verfahren wird im Server ein einziger Prozeß gestartet, in dem mehrere Serverobjekte existieren können. Der Objektadapter kann allerdings nur einen Aufruf des Servers weiterleiten, weitere Aufrufe müssen warten.

- **Unshared Server**
  Im Gegensatz zum *shared server*, wird hier für jedes Objekt vom Objektadapter ein Prozeß gestartet, sobald eine Anfrage an ein Objekt gestellt wird, für das noch kein Prozeß existiert. Auch nachdem der vom Client gewünschte Dienst erbracht wurde, bleibt ein Prozeß aktiv (es sei denn, er wird vom Objektadapter explizit deaktiviert).

- **Server per method**
  Hier wird immer dann ein neuer Prozeß für ein Objekt gestartet, wenn eine entsprechende Anfrage erfolgt. Der Prozeß ist nur für die Dauer der Objektausführung aktiv.

- **Persistent Server**
  Beim *persistent server* liegt es nicht im Aufgabenbereich des Objektadapters, Prozesse für Objekte zu aktivieren oder zu deaktivieren. Zwischen Objektadapter und Serverprozesse ist ein Scheduler geschaltet, der die Prozeßverwaltung übernimmt. Der Objektadapter kommuniziert nur mit dem Scheduler.

Objektimplementierungen können mit verschiedenen ORBs eingesetzt werden. Voraussetzung ist, daß der jeweilige ORB das sog. *language mapping* zur Implementierungssprache des Objektes unterstützt.

Mit Hilfe der IDL werden Objekttypen festgelegt. Dies geschieht durch Angabe der Objektschnittstelle. Diese *Objektschnittstelle* besteht aus einer Menge von benannten Operationen mit deren Ein- und Ausgabeparametern.

2.4 CORBAservices

Die Operationen, die von CORBAservices zur Verfügung gestellt werden können, umfassen:

- **Class Management** (Klassenmanagement)
  Die Möglichkeit, Klassendefinitionen, Klassenschnittstellen und Beziehungen zwischen Klassendefinitionen zu erzeugen, ändern, löschen, kopieren, verteilen, beschreiben und zu kontrollieren.

- **Instance Management** (Instanzenmanagement)
  Die Möglichkeit, Objekte und Objektbeziehungen zu erzeugen, verändern, löschen, kopieren, verschieben, aufrufen und zu kontrollieren.

---

[1] Unter einer Objektreferenz kann man die Informationen verstehen, die benötigt werden, um innerhalb des ORB ein Objekt zu bestimmen

- **Storage** (Speicherung)
  Ein Dienst für die permanente oder temporäre Speicherung von großen oder kleinen Objekten einschließlich ihres Zustandes und ihrer Methoden.

- **Integrity** (Integrität)
  Die Möglichkeit, Konsistenz und Integrität von Objektzuständen sowohl innerhalb einzelner Objekte (z.B. durch *locks*) als auch zwischen Objekten (z.B. duch Transaktionen).

- **Security** (Sicherheit)
  Die Möglichkeit Zugriffsbeschränkungen in einem angemessenem Maß an Granularität auf Objekte und ihre Komponenten einzurichten (d.h. zu definieren und zu erzwingen).

- **Query** (Abfragen)
  Die Möglichkeit, Objekte oder Klassen durch Angabe ihrer Eigenschaften aus implizit oder explizit bestimmten Mengen auszuwählen.

- **Versions** (Versionsmanagement)
  Die Möglichkeit, Objektversionen zu speichern, abzustimmen und zu managen.

Oben genannte CORBAservices müssen jedem OMG-konformen Produkt enthalten sein. Dies ist bei CORBAfacilities (s.u.) nicht der Fall.

2.5 CORBAfacilities

Verglichen mit den CORBAservices versteht man unter CORBAfacilities höherwertige Dienste. Diese sind für ein OMG-konformes Produkt auch nicht fest vorgeschrieben (wie die CORBAservices) sondern optional. CORBAfacilities haben den Charakter einer allgemein benutzbaren Klassenbibliothek. So können sich die verschiedensten Anwendungen der angebotenen Dienste bedienen. Durch den objektorientierten Ansatz ist es zudem möglich, durch Subklassenbildung (und der damit einhergehenden Spezialisierung) schnell zu individuellen Anwendungslösungen zu kommen.

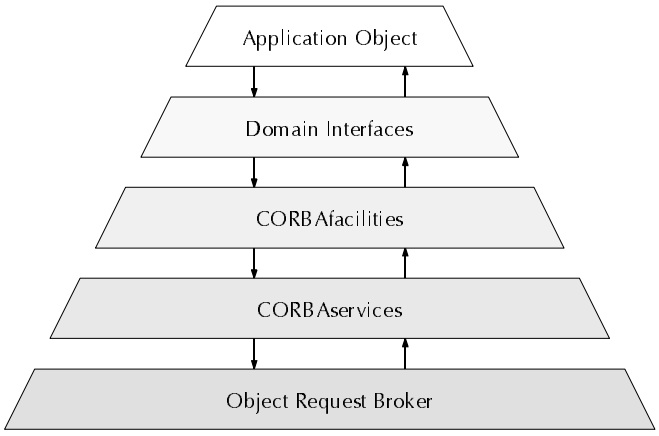


Abbildung 5: Einordnung der CORBAfacilities in die OMA

Folgende Liste enthält eine (keinesfalls vollständige) Aufstellung von Diensten, die als CORBAfacilities in Frage kommen:

- wiederverwendbare Benutzerschnittstellen (z.B. für Texteditoren)
- Druckdienste (z.B. Druckspooler)
- E-Mail-Dienste
- Hilfefunktionen
- Agentfunktionen (z.B. für Systemmanagementagenten)

2.6 Application Objects

Im Gegensatz zu CORBAfacilities (siehe unten) sind Application Objects (AO) nicht von der OMG standardisiert. AOs stellen eine auf eine speziellen Anwendungsfall zugeschnittene Funktionalität bereit, sind also nicht von einem „Allgemeinnutzen“ wie die CORBAfacilities.

Zu Programmen, die möglicherweise in die AO-Klassifikation fallen gehören unter anderem:

- Büroanwendungen: Textverarbeitung, Tabellenkalkulationen, Email etc.
- CAD-Anwendungen
- CASE-Tools
- Netz- und Systemmanagementanwendungen etc.

2.7 SOM/DSOM

2.7.1 SOM als „Objektmanager“

Mit dem System Object Model (SOM) von IBM steht eine CORBA 1.1-konforme, von Programmiersprachen unabhängige[2] Umgebung zur Verfügung, mit der es möglich ist, Klassenbibliotheken zu entwickeln und bereitzustellen. SOM ist also keine Programmiersprache, sondern eher eine Art „Objektmanager“. Das System besteht aus mehreren Komponenten:

- **CORBA-konformer IDL-Compiler**
  mit diesem Compiler ist es möglich, Klassen in IDL zu deklarieren, und in jeder gewünschten Programmiersprache zu implementieren. Der Compiler erzeugt dabei aus den IDL-Schnittstellenbeschreibungen der öffentlichen Klassenschnittstellen Client IDL Stubs, Server Interface Stubs, Implementierungsdateien, die „Codegerüste“ enthalten, die mit dem eigentlichen Implementierungscode aufgefüllt werden können sowie Importdateien für das Interface Repository (s.u).

- **Interface Repository**
  wie in jedem CORBA-System werden hier die Schnittstellen der im System verfügbaren Objekte gespeichert.

- **System zur Zustellung von Methodenaufrufen**
  Zu den Aufgaben dieses Systems gehört es, das Objekt zu lokalisieren, zu dem die aufgerufene Methode gehört (dabei müssen OO-Konzepte wie Polymorphismus und Vererbung berücksichtigt werden), den Aufruf weiterzuleiten und dann die Ergebnisse an den Aufrufer zurückzuliefern.

2.7.2 Das SOM-Klassenkonzept

Das SOM Laufzeitsystem besteht aus drei Hauptklassen sowie einem Klassenmanager, der Klassen lädt und registriert. Die drei Hauptklassen sind

- **SOMObject**
  Von dieser Klasse stammen alle SOM-Objekte ab.

- **SOMClass**
  Diese Klasse stellt all die Methoden bereit, die benötigt werden, um neue Objektinstanzen zu erzeugen. Sie ist die Hauptklasse, von der alle *Metaklassen* (s.u.) abgeleitet werden. Metaklassen sind dabei spezielle Klassen, die die Objekte (also Instanzen) einer zugehöri-

[2] „Von Programmiersprachen unabhängig“ soll hier heißen, daß Objekte, die in einer bestimmten Programmiersprache entwickelt sind, von Objekten oder Anwendungen aufgerufen werden können, die in einer beliebigen anderen Programmiersprache geschrieben sind.

gen „normalen“ Klasse verwalten. Zu den Funktionen einer Metaklasse gehören neben Instanziierung (Erzeugung eines Objekts) der Klasse alle Operationen, die alle Objekte der Klasse betreffen (z.B. Auflistung der Objekte).

- **SOMClassMgr**
  Von dieser Klasse wird zum Systemstart nur das Objekt SOMClassMgrObject instantiiert. Dieses Objekt hat die Aufgabe, alle im System verfügbaren Klassen zu registrieren.

Wichtig beim SOM-Klassenkonzept ist die Tatsache, daß Klassen wie Objekte behandelt werden, sobald sie zu Laufzeit (nach erstem Aufruf einer Methode) vom Klassenmanager geladen wurden, d.h. von einer Klassendefinition (welche zur Übersetzungszeit erzeugt wird) wird zur Laufzeit ein **SOM Klassenobjekt** instantiiert.

Das Konzept wird in *Abbildung 6* veranschaulicht. Oben sind die Hauptklassen SOMObject und SOMClass zu sehen. Von SOMClass wird zunächst die Metaklasse „Fahrzeugfabrik“ abgeleitet. Anschaulich formuliert: „Die Klasse Fahrzeugfabrik enthält Informationen darüber, wie Objekte der Klasse ‘Fahrzeug’ generiert werden“ (daher die Verbindunglinie von „Fahrzeug“ zur Metaklasse „Fahrzeugfabrik“). Analog wird die Klasse „Autofabrik“ von der Klasse „Fahrzeugfabrik“ abgeleitet. „Autofabrik“ verwaltet Objekte der Klassen „Auto“, „Rolls Royce“ und „Käfer“.

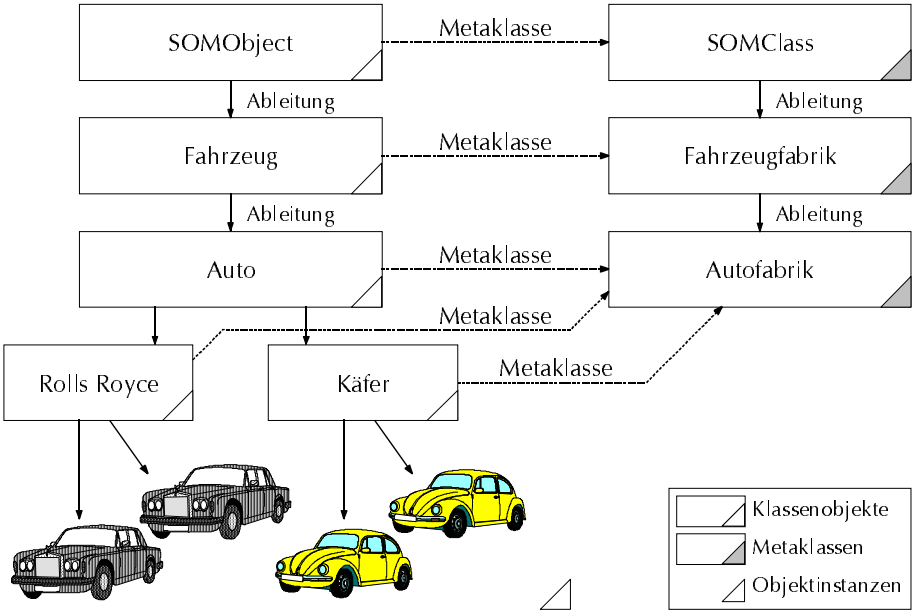


Abbildung 6: SOM Klassen, Metaklassen und Objekte

2.7.3 Distributed SOM (DSOM)

Das bisher beschriebene SOM regelt die Interaktion von Objekten, die in einem gemeinsamen Prozess existieren. SOM stellt also einen implementierungsunabhängigen Rahmen für Objekte zur Verfügung.

Distributed SOM (DSOM) erweitert dieses Konzept nun um Unabhängigkeit vom Ort, an dem sich Objektaufrufer und das aufgerufene Objekt befinden. In DSOM ist es egal, ob sich Aufrufer und Aufgerufener in verschiedenen Prozessen (*Workstation DSOM*) oder gar auf verschiedenen Rechnern (*Workgroup DSOM*) in einem Netz befinden.

2.8 Generelle Entwicklung und Bereitstellung eines CORBA-Servers

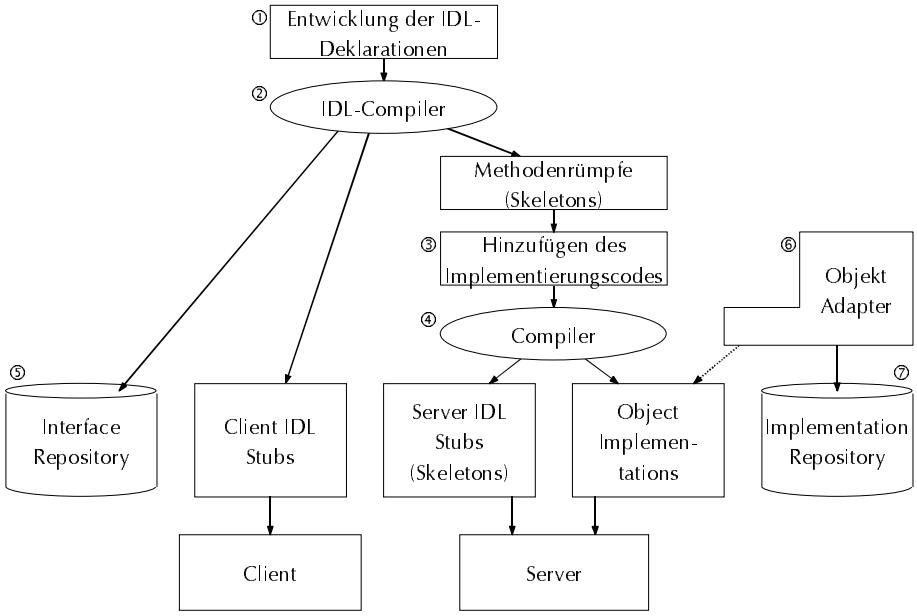


Abbildung 7: Entwicklungsschritte für CORBA Dienste – von der IDL-Beschreibung zu den Interface-Stubs

Es sind also folgende Schritte zu unternehmen:

1. **Deklaration der Objektklassen mit Hilfe der IDL**
   Objekttypen, ihre Attribute, die exportierten Methoden sowie deren Methodenparameter müssen in IDL beschrieben werden. Hier wird kein Implementierungscode programmiert.

2. **Vorcompilieren der IDL-Datei**
   Der IDL-Compiler erzeugt aus der IDL-Datei typischerweise C-Skeletons, die im nächsten Schritt dann mit Implementierungscode aufgefüllt werden. Anstelle von C ist allerdings jede Sprache denkbar, für die ein sog. *language mapping* zur IDL existiert. Neben den Skeletons werden auch **Client Stubs** generiert, die, in Clients eingebunden, diesen den statischen Aufruf der Serverobjekte ermöglichen

3. **Hinzufügen von Implementierungscode**
   An dieser Stelle werden die Serverklassen implementiert, d.h. es wird Code programmiert, der die durch die IDL-Schnittstellen beschriebene Funktionalität erbringt.

4. **Übersetzen des Codes**
   Ein CORBA-konformer Compiler sollte in der Lage sein, folgende drei Typen von Ausgabedateien zu erzeugen: 1.) **Importdateien**, die in das *Interface Repository* eingebunden werden können; 2.) **Server Stubs**, die es dem Objektadapter ermöglichen, die Serverobjekte aufzurufen bzw. die Objekte im *Implementation Repository* zu registrieren; 3.) den eigentlichen **Programmcode** für die Objekte.

5. **Einbinden der Klassendeklarationen in das Interface Repository**
   IDL-Deklarationen werden dem Interface Repository hinzugefügt, so daß sich Anwendungen zur Laufzeit über die Schnittstellen der verfügbaren Objekte informieren können.

6. **Instantiieren der Objekte auf dem Server**
   Bevor die Objekte „ihre Arbeit aufnehmen“ können, müssen sie zuerst vom Objektadapter instantiiert werden. Hierfür gibt es mehrere Strategien: *shared server*, *unshared server*, *server-per-method*, und *persistent server* (näheres hierzu siehe Kapitel 2.3.3).

7. **Registrieren der Laufzeitobjekte im Implementation Repository**
   Der Objektadapter speichert im *Implementation Repository* eine Objektreferenz und den Typ eines jeden Objektes, das er im Server instantiiert.

3 Gewinnung eines geeigneten Objektmodells

3.1 Konkretes Vorgehen unter Verwendung der OMT

3.1.1 Die Object Modeling Technique (OMT)

Die Object Modeling Technique (OMT) nach Rumbaugh sieht folgende Schritte bei der objektorientierten Entwicklung eines Softwaresystems vor:

1. **Analyse**
   In der Analysephase sollen Kandidaten ermittelt werden, die später im System Objekte werden. Inzwischen von Rumbaugh in die erweiterte OMT („*unified method*“) übernommen ist dabei das Konzept sogenannter „Use Cases“ (Fallstudien). Eine kurze Beschreibung der Analysephase des vorliegenden Projektes findet sich in *Kapitel 3.2*.

2. **Design**
   Schrittweise Entwicklung der implementierungsorientierten Konzeption des Softwaresystems aus der problemorientierten Struktur des Problems. In der Designphase wird mit drei Modellen gearbeitet: dem Objektmodell (beschreibt die statische Struktur von Objekten), dem Dynamikmodell (beschreibt dynamische Veränderungen des Systems) und dem Funktionsmodell (beschreibt die Datenänderungen des Systems). Die Designphase der vorliegenden Arbeit wird in *Kapitel 4* beschrieben

3. **Implementierung**
   Übertragung des Objektdesigns in eine reale Programmiersprache, Datenbank oder Hardwareimplementierung.

3.1.2 Werkzeuge zur Unterstützung der Modellierung

Die objektorientierte Modellierung der MIB wurde mit dem Softwarepaket „Software through Pictures“ der Firma Interactive Development Environments durchgeführt. Die OMT-Version von „Software through Pictures“ ist ein Mehrbenutzer-CASE-Tool, das das Zeichnen der verschiedenen OMT-Modelle und das Navigieren zwischen den verschiedenen Systemsichten komfortabel unterstützt.

Eine genauere Beschreibung dieses CASE-Tools findet sich in *Kapitel 5: „Arbeiten mit dem StP/OMT-Softwarepaket“*.

3.2 Analyse

Der Analyse eines Agentensystems eines Rechners wurde durch die schon bestehende, in ASN.1 beschriebene, SNMP-MIB ein grundsätzlicher Rahmen vorgegeben. Ziel war es, bestehende Variablen und Strukturen, ggf. in optimierter Form, auch im objektorientierten Agentensystem bereitzustellen.

Die Übertragung von der bestehenden MIB in eine objektorientierte Form sollte dabei keinesfalls auf rein syntaktischer Ebene (algorithmisch) erfolgen. Die Vorteile der Objektorientiertheit sollten dabei ebenso Berücksichtigung finden wie die Erfordernisse des Agenteneinsatzes.

Von Jacobson vorgeschlagen und inzwischen von Rumbaugh übernommen ist in der OMT für den Analyseprozeß die Verwendung sogenannter „Use Cases“ vorgesehen. Use Cases sind Anwendungsfälle, in denen das zu entwickelnde System eingesetzt werden soll.

Begonnen wurde hierbei mit der Entwicklung der Anwendungsfälle aus den Anforderungsspezifikationen. Diese wurden im vorliegenden Fall aus den funktionalen Dimensionen des Netz- und Systemmanagements gewonnen:

- Sicherheitsmanagement
- Abrechnungsmanagement
- Leistungsmanagement
- Fehlermanagement
- Konfigurationsmanagement

Nächster Schritt war die Identifikation intern ablaufender Vorgänge (Use Cases) sowie der von außen auf das System (hier den Agenten) einwirkenden Benutzer (sogenannter „Actors"). Die Benutzer sind die jeweiligen „Manager". Ein Vorgang ist zumeist die Manipulation der MIB-Daten entsprechend der jeweiligen Zielsetzung (z.B. Auslesen und Setzen der Variablenwerte oder Ansteuerung einzelner Systemkomponenten).

Abbildung 8 zeigt ein Beispiel eines Use Case Modells für die Aufgabenstellung „Systemmanagement". Das Modell enthält zwei Benutzer (Actors): den Konfigurationsmanager sowie den Fehlermanager. Beide greifen auf das zu managende System zu. In einem Anwendungsfall richtet der Konfigurationsmanager einen neuen Benutzer (User) ein, in einem anderen überprüft der Fehlermanager den Status des Druckers.

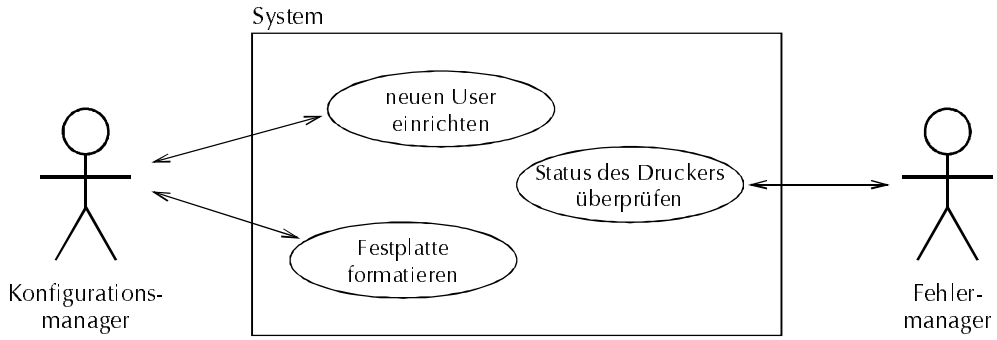


Abbildung 8: Ausschnitt aus der Use Case Modell

Das StP/OMT-Werkzeug unterstützt diesen Analyseprozeß durch einen entsprechenden Use Case Editor, mit dem es möglich ist, ein Use Case Diagramm der obigen Form zu erstellen.

4 Design und Implementierung

4.1 Die Rolle der Verschiedenen Designmodelle

4.1.1 Das Objektmodell

Das Objektmodell war im vorliegenden Fall das mit Abstand wichtigste Modell. Zum einen sollte dieses Modell aus der bestehenden MIB gewonnen werden (also alle wesentlichen Elemente ebenfalls enthalten), andererseits das zu managende System mit objektorientierten Mitteln möglichst realitätsnah beschreiben.

Die Vorgehensweise der Erstellung eines Objektmodells im Rahmen dieses Fortgeschrittenenpraktikums war es, zunächst eine nahezu algorithmische Übersetzung der ASN.1-MIB in ein Objektmodell durchzuführen, welches im nächsten Schritt optimiert wurde. Die Übersetzung der ASN.1-MIB in die Objekt-MIB wird detailliert in *Kapitel 4.2* beschrieben. Mit der anschließenden Optimierung befaßt sich *Kapitel 4.3* und *Kapitel 4.4*.

4.1.2 Das Dynamikmodell

Da die Kontrollstrukturen beim Systemagenten sehr einfach sind, spielte in diesem Fall das Dynamikmodell in der Designphase kaum eine Rolle. Die meisten Funktionen, die für den Systemagenten vorstellbar sind, fallen in eine der folgenden Kategorien:

- liefere einen Variablenwert (bzw. den Status einer Systemkomponente)
- setze einen Variablenwert (bzw. den Status einer Systemkomponente)
- erzeuge, lösche oder ändere ein Objekt der Klasse XY

Zustandsänderungen finden also so gut wie nicht statt. Deshalb wurde hier auf die Erzeugung eines Dynamikmodells verzichtet.

4.1.3 Das Funktionsmodell

Im vorliegenden Fall spielte das Funktionsmodell ebenfalls eine untergeordnete Rolle. Ähnlich einer Datenbank hält der Systemagent meist nur Informationen, d.h. er speichert sie bzw. stellt sie bereit. Interaktionen zwischen den einzelnen Objekten finden kaum statt. Änderungen während des Einsatzes des Systemagenten bleiben meist lokal beschränkt. Die Funktionalität ist eher trivial. Wegen des nicht zu erwartenden Nutzen wurde deshalb im Rahmen dieses Praktikums kein Funktionsmodells erstellt.

4.2 „Naive“ Übersetzung der SNMP-MIB in ein Objektmodell

In diesem ersten Schritt wurde die bestehende MIB ohne jegliche Optimierung (z.B. noch keinerlei Vererbungsstrukturen) in ein Objektmodell übertragen. Die Übertragung erfolgte dabei auf syntaktischer Ebene gemäß folgenden beiden Regeln:

1. **Jede Gruppe in der LRZ-MIB wird in eine Klasse überführt.**
2. **Eine Tabellenzeilen wird jeweils ein Objekt einer Tabellenklasse, welche mit einer übergeordneten[3] Klasse in 1:n-Beziehung steht. Einfache Variablen einer Gruppe werden in eine eigene Klasse (die dann nur eine Instanz besitzt) verlegt.**

[3] „übergeordnet“ soll hier nicht im Sinne einer Vererbungshierarchie verstanden werden. Eine als übergeordnet bezeichnete Klasse dient als eine Art Container, über die die „untergeordneten“ Tabellenzeilenobjekte verwaltet werden, d.h. erzeugt, gelöscht, geändert werden.

Die erste Regel ist recht naheliegend. Durch die Gruppenbildung in der ASN.1-MIB wurde eine Trennung der verschiedenen Komponenten eingeführt, die in der Objekt-MIB zunächst beibehalten werden sollte. Die zweite Regel geht auf die „Dynamik" von MIB-Tabellen ein (d.h. daß Tabellen zeilenweise wachsen und schrumpfen können, Tabellenzeilen dabei jeweils die gleiche Struktur aufweisen).

Mit Hilfe dieser beiden Regeln war eine schnelle, „mechanische" Übersetzung der ASN.1-MIB in ein Objektmodell möglich. Die Prinzipien der Objektorientierung wurden in diesem Modell aber nur in leichten Ansätzen realisiert.

Im folgenden soll die Übersetzung exemplarisch an einem Ausschnitt der LRZ-MIB verdeutlicht werden. Zunächst folgt ein um die für dieses Beispiel unwesentlichen Abschnitte gekürzter Teil der ASN.1-MIB, danach der entsprechende Ausschnitt der daraus gewonnenen Objekt-MIB:

Hier zunächst die Systemgruppe. Weggelassen sind die Felder **STATUS** (hier ist der Wert immer „**mandatory**") sowie **DESCRIPTION** (hier sollte die Semantik aus dem Variablenbezeichner ersichtlich sein).

```
-- The UNIX-LRZ System Group
-- Implementation of the System Group is mandatory for all
-- systems. If an agent is not configured to have a value for
-- any of these variables, a string of length 0 is returned.

sysName OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS   read-only
     ::={ system 1 }

sysContact OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS read-write
     ::={ system 2 }

sysLocation OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS read-write
     ::={ system 3 }

sysOs OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS read-only
     ::={ system 4 }

sysHardware OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS read-only
     ::={ system 5 }

sysUptime OBJECT-TYPE
     SYNTAX   INTEGER
     ACCESS read-only
     ::={ system 6 }

sysDate OBJECT-TYPE
     SYNTAX   DateAndTime
     ACCESS read-write
     ::={ system 7 }

sysUsers OBJECT-TYPE
     SYNTAX   INTEGER
     ACCESS read-only
     ::={ system 8 }

sysClockTicks OBJECT-TYPE
     SYNTAX   INTEGER
     ACCESS read-only
     ::={ system 9 }

[gekürzt um die status- und description-Felder]
```

Gemäß Regel 1 wurde also aus der gesamten Gruppe eine Klasse, die sämtliche Variablen der ASN.1-MIB als Attribute enthält. Die einzelnen Variablentypen wurden weitgehend ohne Änderung übernommen. Ausnahme ist lediglich der Datentyp **Display-String**. Dieser erscheint im

Objektmodell durchgehend als **`string`**, der mit `""` (dem Leerstring) initialisiert wird. Danach ergab sich die folgende Klasse, welche in *Abbildung 9* zu sehen ist.

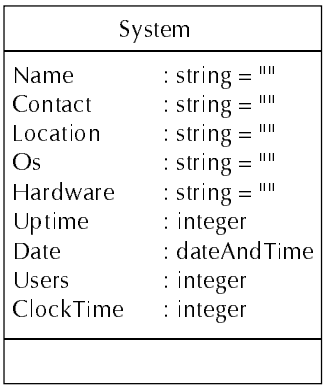


Abbildung 9: Die Klasse „System“

Das nächste Beispiel beschreibt die Übersetzung einer MIB-Gruppe, die eine Tabelle enthält: die Speichergruppe (**`Storage Group`**). Im folgenden der entsprechende ASN.1-Code. Dieser wurde gekürzt um die Variablendefinitionen der Tabelle **`stoTable`**. Der Aufbau einer Tabellenzeile, also die Tabellenspalten sind an der Struktur von **`stoEntry`** ablesbar.

```
-- The UNIX-LRZ Storage Group
-- Implementation of the Storage Group is mandatory for all
-- systems. If an agent is not configured to have a value for
-- any of these variables, a string of length 0 is returned.

stoTypes          OBJECT IDENTIFIER ::= { storage 1 }
stoRam            OBJECT IDENTIFIER ::= { stoTypes 1 }
stoVirtualMemory  OBJECT IDENTIFIER ::= { stoTypes 2 }
stoFLCache        OBJECT IDENTIFIER ::= { stoTypes 3 }
stoSLCache        OBJECT IDENTIFIER ::= { stoTypes 4 }

stoTable OBJECT-TYPE
     SYNTAX SEQUENCE OF StoEntry
     ACCESS not-accessible
     ::= { storage 2 }

stoEntry OBJECT-TYPE
     SYNTAX       StoEntry
     ACCESS       not-accessible
     INDEX { stoIndex }
     ::= { stoTable 1 }

stoEntry ::= SEQUENCE {
     stoIndex              INTEGER,
     stoType               OBJECT IDENTIFIER,
     stoDescr              DisplayString,
     stoAllocationUnits    Bytes,
     stoSize               KBytes,
     stoUsed               INTEGER,
     stoState              INTEGER
     }

[gekürzt um die stoEntry-Variablendefinitionen]
```

Zunächst wurde die Speichergruppe, wie in Regel 1 verlangt, in ein Objekt übertragen. Dieser Vorgang entsprach auch der Forderung in Regel 2, da die gesamte Speichergruppe nur aus einer Tabelle besteht. Das heißt also, daß es keine für alle Speicher gemeinsamen MIB-Variable gibt. Als übergeordnete Klasse diente hier die oben beschriebene Systemklasse. Die Speicherklasse sieht demnach wie folgt aus (Ein Verzeichnis der OMT-Symbole, die in dieser Arbeit verwendet wurden findet sich in *Anhang B: Teil der OMT-Notation nach* Rumbaugh)

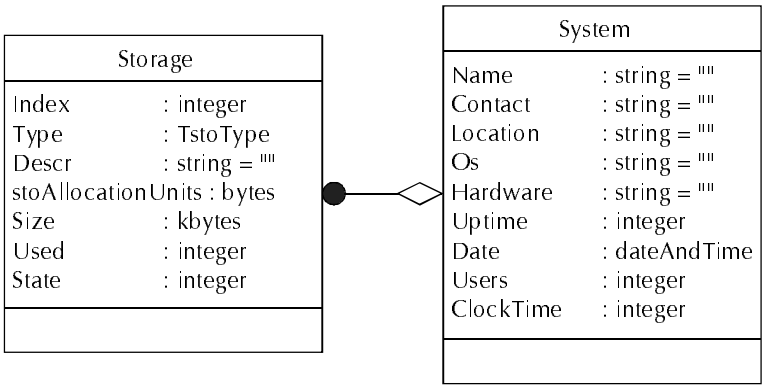


Abbildung 10: Die Klasse „Storage“ mit der ihr übergeordneten Systemklasse

Im Gegensatz zur Speichergruppe, die aus einer einzigen Tabelle besteht, gibt es in der Prozessorgruppe MIB-Variablen, die zu einer Tabelle gehören, aber auch Variablen, die sich auf alle Prozessoren beziehen. Im folgenden der Ausschnitt aus der LRZ-MIB, der wiederum gekürzt ist um den Status und die Beschreibung der einzelnen Variablen sowie um einige Definitionen der Tabellenvariablen:

```
-- The UNIX-LRZ Processor Group
-- Implementation of the Processor Group is mandatory for all
-- systems. If an agent is not configured to have a value for
-- any of these variables, a string of length 0 is returned.

cpuType OBJECT-TYPE
     SYNTAX   DisplayString
     ACCESS read-only
     ::={ processor 1 }

cpuClockRate OBJECT-TYPE
     SYNTAX   INTEGER
     ACCESS read-only
     ::={ processor 2 }

cpuTable OBJECT-TYPE
     SYNTAX   SEQUENCE OF CpuEntry
     ACCESS not-accessible
     ::={ processor 3 }

cpuEntry OBJECT-TYPE
     SYNTAXcpuEntry
     ACCESS   not-accessible
     INDEX    { cpuNr }
     ::={ cpuTable 1 }

cpuEntry ::= SEQUENCE {
     cpuNr              INTEGER,
     cpuOpStat          INTEGER,
     cpuUsStat          INTEGER,
     cpuAdStat          INTEGER,
     cpuAction          INTEGER,
     cpuUserTime        INTEGER,
     cpuNiceTime        INTEGER,
     cpuSystemTime      INTEGER,
     cpuIdleTime        INTEGER
     }

[...]

cpuOpStat OBJECT-TYPE
     SYNTAX   INTEGER {
     enabled(1),
     disabled(2)
     }
     ACCESS read-only
     ::={ cpuEntry 2 }
```

```
cpuUsStat OBJECT-TYPE
     SYNTAX   INTEGER{
     idle(1),
     busy(2)
     }
     ACCESS read-only
     ::={ cpuEntry 3 }

cpuAdStat OBJECT-TYPE
     SYNTAX   INTEGER {
     unlocked(1),
     locked(2),
     shuttingdown(3)
     }
     ACCESS read-only
     ::={ cpuEntry 4 }

cpuAction OBJECT-TYPE
     SYNTAX   INTEGER {
     enable(1),
     disable(2),
     lock(3),
     unlock(4),
     shuttingdown(5)
     }
     ACCESS read-write
     ::={ cpuEntry 5 }

[gekürzt um weitere Variablenbeschreibungen]
```

In diesem Fall mußte also gemäß Regel 2 eine Klasse **Cpu** für die Tabellenzeilen eingeführt werden. Als übergeordnete Klasse diente hier wieder die Systemklasse. Für die Variablen, die außerhalb der Tabelle liegen, also für alle Prozessoren gelten, wurde eine eigene Klasse **CpuGeneral** definiert. Damit sind die Prozessorklassen wie folgt definiert:

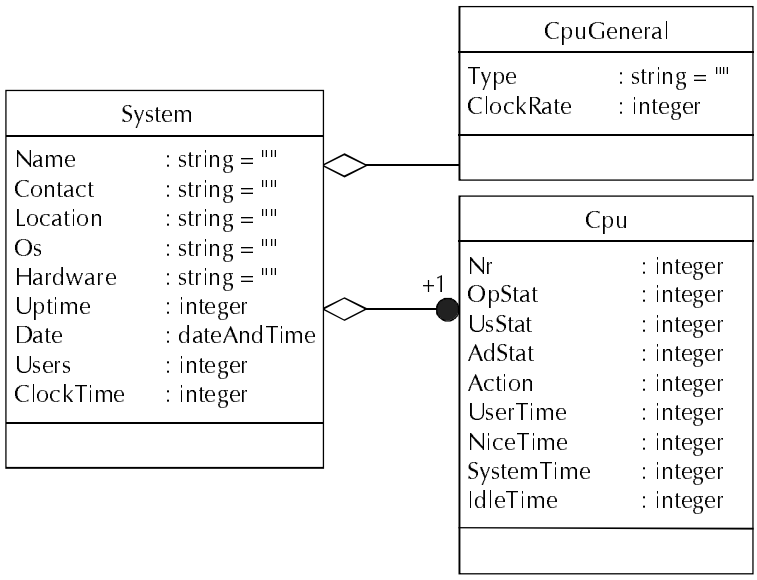


Abbildung 11: Die beiden Prozessorklassen „Cpu“ und „CpuGeneral“ mit der ihnen übergeordneten Systemklasse

Die Tatsache, daß es von der Klasse **Cpu** viele (aber mindestens eine) Instanzen geben kann, wurde im Objektmodell durch ein entsprechendes Symbol[4] (der ausgefüllte Kreis mit dem Bezeichner „+1“) wiedergegeben. Dagegen fehlt dieser Bezeichner natürlich bei der Klasse **CpuGeneral**, da hier ja nur eine Instanz benötigt wird.

[4] Ein Verzeichnis der OMT-Symbole, die in diesem Praktikum verwendet wurden, findet sich in *Anhang B: Teil der OMT-Notation nach Rumbaugh*

Abbildung 12 zeigt nun den Ausschnitt der ersten Objekt-MIB, die alle Klassen umfaßt, die bisher hier beschrieben wurden. Hinzu kommt die Klasse **`Device`**, die aus der **`Device`**-Gruppe abgeleitet wurde. Die Erstellung dieser Klasse erfolgte dabei analog der **`Storage`**-Klasse, da die **`Device`**-Gruppe ebenfalls eine einzige Tabelle umfaßt. Eine vollständige Abbildung des ersten Objektmodells findet sich im Anhang.

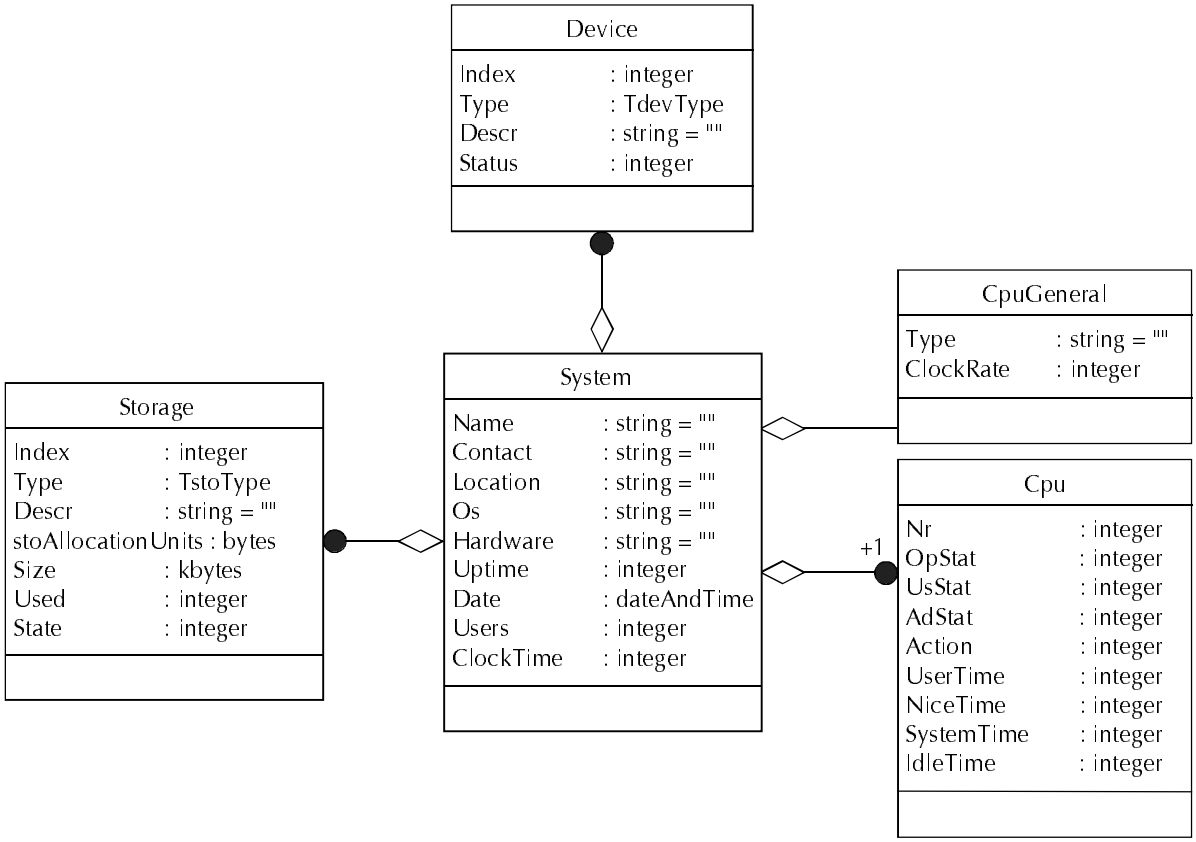


Abbildung 12: Ausschnitt aus der ersten Objekt-MIB

4.3 Bewertung der ersten Objekt-MIB

Nun war mit der bisherigen Übertragung der ASN.1-MIB nach den beiden obigen Regeln in eine Objekt-MIB noch nicht viel gewonnen. Durch die größtenteils „syntaktische" Übersetzung blieben viele Probleme der ASN.1-MIB auch in der Objekt-MIB bestehen. Darüber hinaus wurden objektorientierte Techniken nur minimal berücksichtigt. Hier nun die Kritikpunkte an dieser ersten Version einer objektorientierten MIB im einzelnen:

- **Zwischen den einzelnen Klassen fehlt (bis auf die Enthaltenseinsbeziehung zur Systemklasse) jegliche Hierarchie.**
  Eine Enthaltenseinshierarchie war nur ansatzweise – eben mit der Systemklasse – realisiert, eine Vererbungshierarchie fehlte gänzlich. Damit waren zwei wesentliche Möglichkeiten der Objektorientierung noch nicht ausgeschöpft. Polymorphismus wurde damit zwangsläufig auch nicht unterstützt.

- **Die noch vorhandenen Typvariablen widerstreben objektorientierten Grundsätzen**
  Beispiele für solche Typvariablen in obigen MIB-Ausschnitten sind **`Type`** in der **`Storage`**-Klasse oder **`Type`** in der **`Device`**-Klasse. Damit war es zwar möglich, verschiedene Speichertypen (durch entsprechendes Setzen der Typvariablen) zu instantiieren, die Objektstruktur wäre aber für die einzelnen Speicherobjekte die gleiche gewesen, unterschiedliche Funktionen der verschiedenen Typen wurden also nicht unterstützt.

- **Die Variablentypen beschränken sich größtenteils nur auf ASN.1-Grundtypen**
  Auch Variablen mit stark eingeschränkten Wertebereich, wie z.B. **`OpStat`** oder **`UsStat`**[5] in der Klasse **`Cpu`**,besaßen im Objektmodell einen einfachen **`integer`**-Typ.

- **Die Problematik sogenannter „Pushbutton"-Variablen bleibt bestehen**
  Das Problem, daß die Wertzuweisung an eine Variable unmittelbar eine Operation auslöst, war auch in diesem ersten Objektmodell vorhanden.

Wichtig: **Das objektorientierte Prinzip der Einkapselung ist in diesem Stadium bereits berücksichtigt!**
Wäre diese Objekt-MIB als Basis für die Implementierung herangezogen worden, so hätte der IDL-Compiler Rahmendateien erzeugt, in denen die einzelnen Attribute als **`private`** gekennzeichnet gewesen wären, also nur über spezielle **`get-`**/**`set-`**Operationen (deren Rahmen ebenfalls erzeugt würden) zugreifbar gewesen wären.

4.4 Optimierung des Modells

Aus obigen Beobachtungen ließen sich folgende Regeln für die Optimierung der Objekt-MIB ableiten. Erläuterungen und Beispiele[6] finden sich in den jeweils angegebenen Unterkapiteln:

1. In mehreren Klassen gemeinsame Attribute und Operationen wurden in eine (ggf. neu einzuführende) Superklasse „nach oben gezogen" (→ *Kapitel 4.4.1*).
2. Typenbezeichner wurden entfernt. An Stelle dieser Bezeichner wurden neue Unterklassen eingeführt (→ *Kapitel 4.4.2*).
3. „Pushbutton-Variablen" wurden zu Methoden der jeweiligen Klasse (→ *Kapitel 4.4.3*).
4. Möglichst realitätsgetreue Modellierung von Objektbeziehungen; Einführung von objektorientierten Hierarchien (Vererbung und Enthaltensein) (→ *Kapitel 4.4.4*).
5. Definition neuer Variablentypen für Attribute mit bestimmten Wertebereichen (z.B. Aufzählungstypen) (→ *Kapitel 4.4.5*).

4.4.1 Neue Superklassen für gemeinsame Attribute und Operationen

Auffällig an der ersten Objekt-MIB war, daß mehrere Klassen Attribute mit identischer Bedeutung hatten. Eine solche Situation sollte in Objektmodellen aber möglichst vermieden werden. Anstelle dessen bot es sich an, gemeinsame Attribute in eine Oberklasse aufzunehmen, betreffende Klassen von dieser Oberklasse abzuleiten und dafür in den Unterklassen die gemeinsamen Attribute zu streichen.

Beispiele für gemeinsame Attribute mehrerer Klassen waren ein Identifikator, ein Name und Statusvariablen. Diese traten u. a. im Objekt **`Printer`**, **`Speicher`** und **`Cpu`** auf. Anstatt nun eine neue Oberklasse einzuführen, wurde der Klasse **`Device`** (umbenannt in **`Generic_Device`**) die Rolle der Oberklasse zugewiesen. Insgesamt wurde die Aufgabe der Klasse **`Device`** geändert. Sie diente nun als Wurzel der Vererbungshierarchie vieler Systemkomponenten. Dies erleichtert auch spätere Erweiterungen der MIB, bei denen eine neue Systemkomponente von **`Generic_Device`** abgeleitet werden kann, wodurch schon eine gewisse Grundfunktionalität bereit gestellt werden kann (eben die, die in den meisten Komponenten gleich ist).

---

[5] cpuOpStat kann die Werte 1 (enabled) und 2 (disabled) annehmen – dafür würde der Datentyp boolean ausreichen; cpuAdStat kann die Werte 1 (unlocked), 2 (locked) und 3 (shuttingdown) annehmen – hier würde sich ein Aufzählungstyp anbieten.

[6] Die Beispiele beschränken sich dabei größtenteils auf die Teile der Objekt-MIB, die jeweils den oben gezeigten Objekt-MIB-Ausschnitten entsprechen. Diese Teile der optimierten MIB werden dabei nicht einzeln, sondern in einer Gesamtübersicht (*Abbildung 15*) dargestellt.

4.4.2 Neue Unterklassen anstelle von Typvariablen

Dieser Optimierungsansatz war die Antwort auf den zweiten Kritikpunkt an der ersten Objekt-MIB (*siehe Kapitel 4.3*). Durch das Konzept der Typvariablen war es zwar möglich, Objekte verschiedener Typen einer Klasse zu instantiieren, es war jedoch nicht möglich, verschiedenen Typen verschiedene Eigenschaften (unterschiedliche Attribute, Operationen etc.) zuzuweisen; jedes Objekt der Klasse – gleich welchen Typs – hätte den gleichen Aufbau gehabt.

So konnte man zwar Objekte verschiedenen Typs der Klasse **`Device`** erzeugen (durch entsprechendes Belegen der Variable **`Type`**), für jede dieser Komponenten hätten dann jedoch nur die drei Variablen **`Index`**, **`Desc`** und **`Status`** zur Verfügung gestanden. Dies war ein weiterer Grund, weshalb die **`Generic_Device`**-Klasse zur Wurzel des Vererbungsbaumes gemacht wurde (*siehe auch Abbildung 15*). Soll nun ein Objekt für eine Systemkomponente erzeugt werden, so wird nicht die **`Generic_Device`**-Klasse instantiiert, sondern eine entsprechende Subklasse

Ähnlich verhielt es sich mit der **`Storage`**-Klasse. Hier konnten zwar die Typen **`Ram`**, **`VirtualMemory`**, **`FLCache`** und **`SLCache`** erzeugt werden, es war jedoch nicht möglich, den einzelnen Typen auch unterschiedliche Eigenschaften zuzuordnen. Deshalb wurden die neuen Klassen **`permanentStorageDevice`** und **`volatileStorageDevice`** eingeführt. Dabei fielen unter die erste Subklasse Komponenten wie Festplatten, Tapes etc. Die zweite Subklasse sollte die Komponenten umfassen, die in der ersten MIB über die **`Storage`**-Klasse instantiiert wurden. Als Folge dieser Anforderung wurden die Subklassen **`Ram`**, **`VirtualMemory`**, **`FLCache`** und **`SLCache`** vorgesehen (*siehe Abbildung 15*).

4.4.3 Operationen anstelle von „Pushbutton"-Variablen

Das Problem der sogenannten „Pushbutton"-Variablen, welches in der ASN.1-MIB bestand wurde auch in die erste, „naive" MIB übernommen: Eine Operation auf einem Objekt (und als Folge auf der entsprechenden Systemkomponente) mußte dadurch ausgelöst werden, daß einem entsprechenden Objektattribut ein Wert zugewiesen wurde. Bezüglich der Semantik war dies ein sehr fragwürdiger Weg, da auf den ersten Blick nicht zwischen einem „normalen" Wertattribut und einem „Pushbutton"-Attribut unterschieden werden konnte.

Im objektorientierten Ansatz war dieses Problem jedoch sehr leicht zu lösen: Es wurden einfach Operationen eingeführt, das „Pushbutton"-Attribut entfiel. Für jeden Wert dieses Attributes wurde dabei eine eigene Operation eingeführt. Ein Beispiel hierfür ist das Attribut **`cpuAction`** der Prozessorklasse. Für die fünf Werte, die diesem Attribut zugewiesen werden konnten, wurden die entsprechenden Operationen **`enable()`**, **`disable()`**, **`lock()`**, **`unlock()`** und **`shuttingdown()`** eingeführt. Die bisherige Wertzuweisung **`cpuAction=1`** entspricht nun einem Aufruf der Operation **`enable()`**.

Da diese fünf Operationen in vielen Systemkomponenten ebenfalls auftraten, wurden sie in der optimierten Objekt-MIB in die Klasse **`Generic_Device`** aufgenommen (*siehe Abbildung 15*).

4.4.4 Möglichst realitätsgetreue Modellierung der Objektbeziehungen

Hinsichtlich der Beziehungen zwischen Klassen wies die erste Version der Objekt-MIB das gleiche Defizit auf wie die ASN.1-MIB. Es war keinerlei Vererbungshierarchie vorhanden. Assoziations- bzw. Containmenthierarchien waren in der ASN.1-MIB und in der ersten Objekt-MIB nur ansatzweise vorhanden. Ein Grundgedanke der „objektorientierten Philosophie" ist es jedoch, die reale Welt, d.h. die einzelnen Objekte sowie ihre Beziehung untereinander, möglichst gut abzubilden.

Die Einführung einer Vererbungshierarchie wurde schon in *Kapitel 4.4.1* erwähnt. Damit sollten Fälle der Realität modelliert werden, in denen eine Systemkomponente eine Spezialisierung einer anderen Komponente sind. So ist z.B. ein Prozessor ein spezielles Gerät (**`Device`**).

Als Wurzel des Enthaltenseins- oder Containment-Baumes wurde die Systemklasse bestimmt. Dies entspricht auch der Realität: Ein System ist die Komponente, die andere Komponenten

enthält. Dabei sollte die Enthaltenseinsbeziehung der Systemklasse mit möglichst „spezialisierten“ Klassen, also Klassen, die im Vererbungsbaum ganz unten hängen, bestehen. Damit können die einzelnen Beziehungen individuell gestaltet werden. So benötigt ein System mindestens einen Prozessor (hier also eine 1:n-Beziehung mit n≥1), jedoch kann es beliebig viele „**permanentStorageDevices**“ besitzen (in diesem Fall eine 1:n-Beziehung mit n≥0). Ein Drukker wiederum ist kein wesentlicher Bestandteil eines Systems. Hier besteht also keine Aggregation, sondern eine einfache 1:n-Beziehung mit n≥0. Wäre die Beziehung zwischen System und höherliegenden Klassen modelliert worden, so wäre eine individuelle Gestaltung nicht möglich gewesen. Jede Subklasse hätte die gleiche Beziehung zu System wie die Oberklasse. *Abbildung 13* gibt einen Überblick über die Beziehungsstruktur eines Teils der optimierten Objekt-MIB.

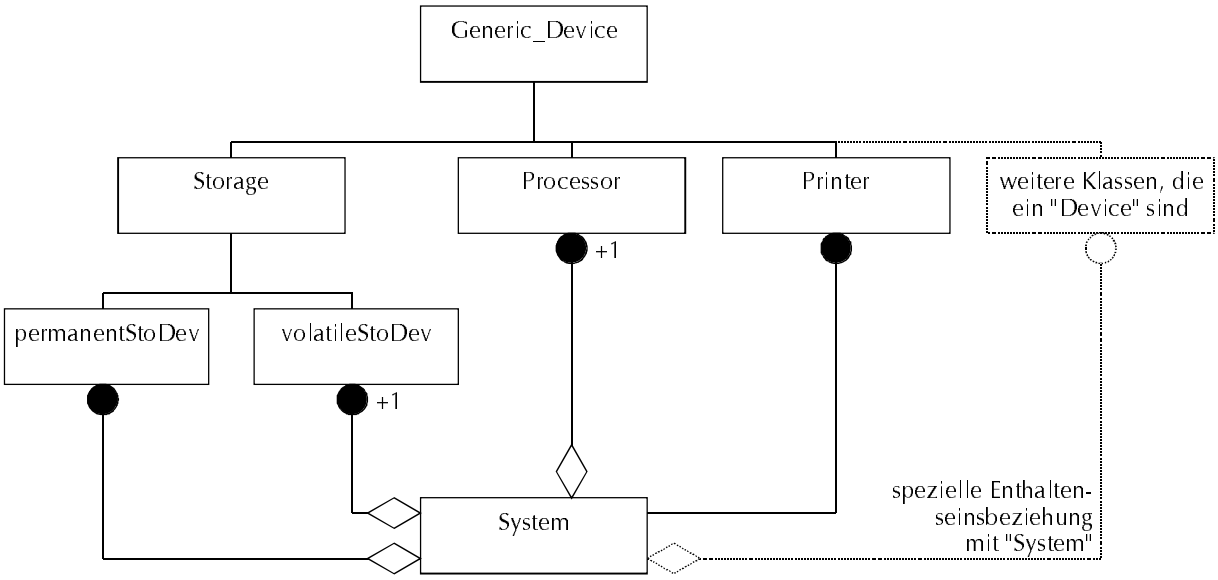


Abbildung 13: Die Beziehungsstruktur in der optimierten Objekt-MIB

Eine weitere Besonderheit findet sich in der Beziehung zwischen **Filesystem** und **Account** (vormals die **User**-Klasse). Account enthielt in der ersten MIB Attribute für Quotas. Dies war semantisch eigentlich nicht korrekt, da eine Quota keine Eigenschaft eines Benutzers ist, sondern genaugenommen eine Eigenschaft der Beziehung zwischen einem Benutzer und einem Filesystem. Aus diesem Grunde wurde die zur Quota gehörenden Attribute ausgelagert und daraus eine Assoziationsklasse modelliert. Diese neue Klasse bezieht sich auf die Beziehung User-Filesystem. Folgende Abbildung veranschaulicht dies:

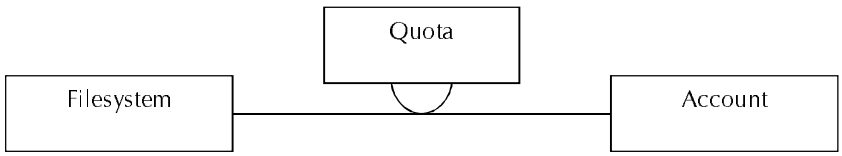


Abbildung 14: „Quota“ als Assoziationsklasse der Beziehung User-Filesystem

4.4.5 Neue Variablentypen für Variablen mit eingeschränktem Wertebereich

Die „mechanische“ Übersetzung der ASN.1-MIB hatte nichts daran geändert, daß die meisten Attributtypen im Objektmodell einfache ASN.1-Grundtypen waren. Dies war in vielen Fällen allerdings eher mangelhaft. So hat die Variable **OpStat** in der Klasse **Cpu** den Typ **integer**, obwohl sie eigentlich nur die Werte 1 (für „enabled“) und 2 (für „disabled“) annehmen konnte. Es wäre nun im Rahmen der Optimierung naheliegend gewesen, den Typ dieser Variablen auf **boolean** zu setzen. Es erwies sich jedoch als besser, einen (gleichwertigen) Aufzählungstypen zu

definieren, der sofort erkennen läßt, in welchem Nutzungszustand sich ein Gerät befindet (eben „enabled“ oder „disabled“).

Ein weiteres Beispiel für die Einführung eines neuen Datentyps ist die Variable **`AdminState`** (vormals **`AdStat`**). Hier wurde (aus entsprechenden Gründen) ein Aufzählungstyp definiert, der die Administrationszustände „unknown“, „unlocked“, „shutting down“ und „locked“ zuläßt.

Wichtig: Soll aus dem Modell (wie in diesem Praktikum) mit StP eine IDL-Ausgabe erzeugt werden, so ist unbedingt darauf zu achten, daß im Klassentabelleneditor für die Attribute und Operationen die gewünschten Einstellungen in den Spalten für IDL-Code gemacht werden. Insbesondere sollten Nur-Lese-Attribute als **`readonly`** gekennzeichnet werden. Andernfalls erzeugt StP Schnittstellenbeschreibungen die aufwendig (bzw. aufwendiger als ohnehin schon; s. u.) nachbearbeitet werden müssen.

Zum Abschluß des Kapitels 4.4 hier nun der zu *Abbildung 12* entsprechende Ausschnitt aus der optimierten Objekt-MIB. Das vollständige optimierte Objektmodell findet sich im Anhang.

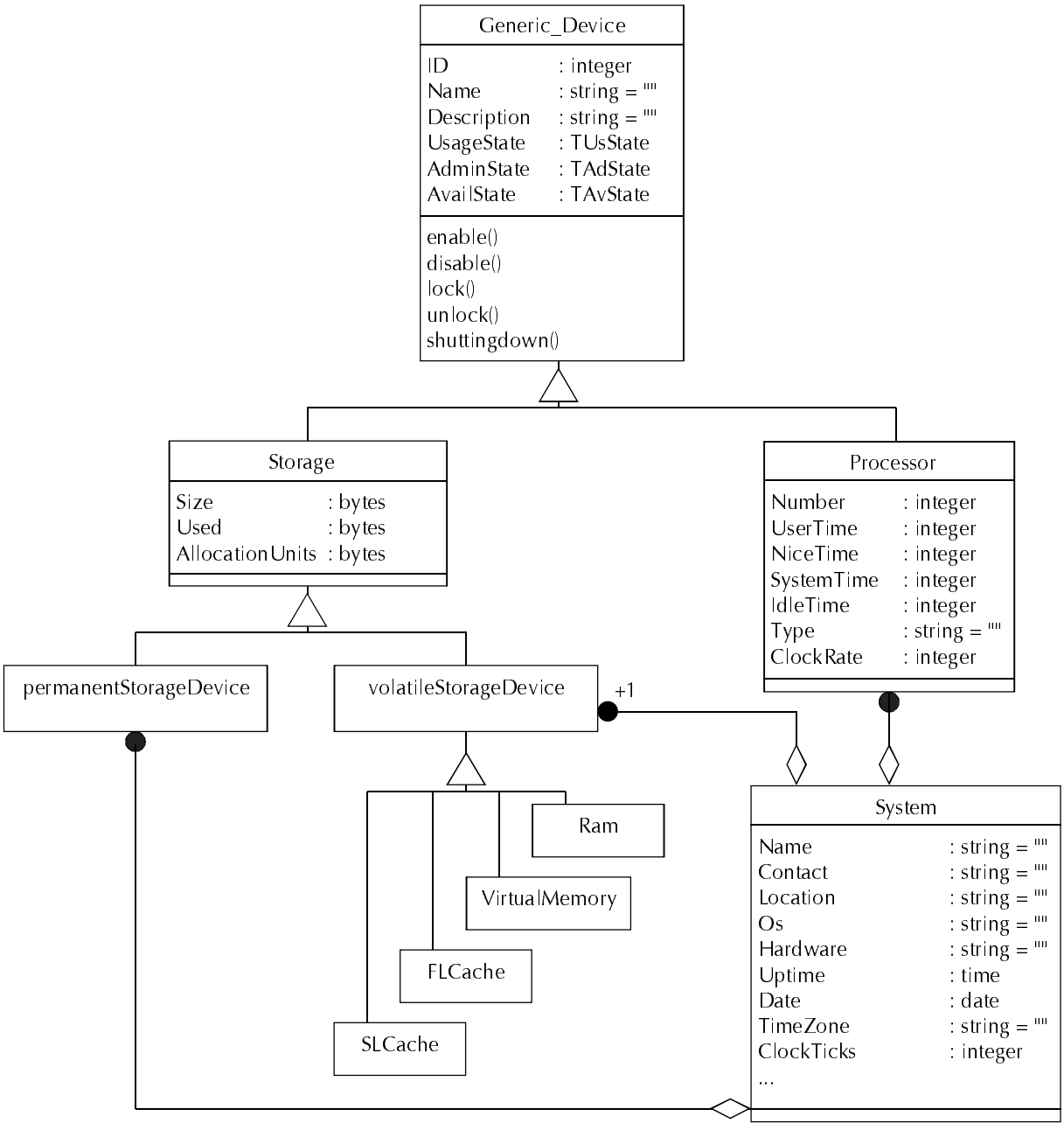


Abbildung 15: Ausschnitt aus der optimierten Objekt-MIB

4.5 Implementierung

4.5.1 Implementierungsbedingte Nachbesserungen des Objektmodells

Nachdem die objektorientierte Modellierung der MIB abgeschlossen war, sollte nun die Implementierung folgen. Dabei stellte sich heraus, daß – entgegen des Gedankens der OMT – kleine, implementierungsbedingte Nachbesserungen der MIB notwendig wurden.

Da als zugrundeliegende Architektur CORBA vorgesehen war, wurden zunächst zum Objektmodell gehörende IDL-Schnittstellenbeschreibungen vom StP-Werkzeug generiert. Dabei stellte sich leider heraus – was eigentlich zu erwarten war –, daß Beziehungen nur mangelhaft in IDL wiedergegeben werden. Als Beispiel dafür die Schnittstellenbeschreibung die für die Systemklasse erzeugt wird:

```
// StP -- created on Mon Apr 29 15:06:21 1996 for ug201db@sun6 from system oo-mib2_02

#ifndef _System_idl_
#define _System_idl_


// stp class declarations
interface System;
// stp class declarations end


// stp class definition 108
interface System
{
// stp class members
    readonly attribute long ClockTicks;
    attribute string Contact;
    attribute date Date;
    attribute string Hardware;
    attribute string Location;
    attribute string Name;
    readonly attribute string Os;
    readonly attribute string TimeZone;
    readonly attribute time Uptime;
    readonly attribute long maxProcessNumber;
    readonly attribute long maxProcessSize;
    readonly attribute long curProcessNumber;
    readonly attribute long curMaxProcessSize;
    readonly attribute long curMaxProcessTime;
    readonly attribute long Avg1;
    readonly attribute long Avg5;
    readonly attribute long Avg15;
    attribute sequence<Printer> assnPrinter;
    attribute Process assnProcess;
    attribute SpecBenchmark assnSpecBenchmark;
    attribute ActiveUser assnActiveUser;
    attribute sequence<Processor> aggrProcessor;
    attribute volatileStorageDevice aggrvolatileStorageDevice;
    attribute sequence<permanentStorageDevice> aggrpermanentStorageDevice;
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Es wird deutlich, daß alle im Klassentabelleneditor gemachten Einstellungen, wie **`readonly`**, Datentypen u.ä., übernommen wurden.

An obiger Ausgabe ist gut zu erkennen, wie Beziehungen aus dem Objektmodell in IDL-Schnittstellenbeschreibungen abgebildet werden. Folgende Tabelle gibt eine Schnellübersicht:

Beziehung...	... zw. System und...	...wird abgebildet in:
1:n-Assoziation	Printer	`attribute sequence<Printer> assnPrinter`
1:1-Assoziation	SpecBenchmark	`attribute SpecBenchmark assnSpecBenchmark`
1:n-Aggregation	Prozessor	`attribute sequence<Processor> aggrProcessor`
1:1-Aggregation	volatileStoDev	`attribute volatileStorageDevice aggrvolatileStorageDevice`

Diese Übersetzung gibt die Beziehungen tatsächlich korrekt wieder. Werden Beziehungen allerdings derart – nämlich als Attribute – implementiert, ergibt sich folgendes Problem: Wird ein Objekt einer Klasse instantiiert, welche in mehreren anderen Klassen durch entsprechende Attribute als assoziiert gekennzeichnet ist, kann es leicht zu Inkonsistenzen kommen, wenn das Objekt z.B. in einem Objekt in der „**`sequence`**“ erfaßt ist, in einem anderen Objekt jedoch nicht.

Bezüglich der Erzeugung und Löschung von Objekten bzw. der Objektverwaltung allgemein ergibt sich ein weiteres Problem: Es besteht keine „zentrale Komponente“, über die Objekte einer Klasse erzeugt, gelöscht oder zum Beispiel aufgelistet werden könnten.

Eine Lösung für dieses und das obige Problem war die Einführung von Metaklassen. D.h. zu jeder Klasse existierte damit eine weitere Klasse, von der allerdings nur ein einziges Objekt instantiiert wird und Funktionen bereitstellt, die der Verwaltung von Objekten der Hauptklasse dienen. Da dieses Konzept ein rein implementierungstechnisches ist, wurde in dieser Arbeit darauf verzichtet, die Metaklassen in das Objektmodell aufzunehmen.

Ein Fall, in dem die Einführung einer noch aus Metaklasse noch aus einem anderen Grund nützlich ist, trat im Zusammenhang mit der Prozeß-Klasse auf. Generell gilt, daß die Objekt-MIB beim Systemstart instantiiert werden sollte. Bei statischen Objekten bzw. Objekten, die sich nur selten ändern, wie z.B. Prozessoren, Festplatten o.ä. ist dies auch kein Problem. Anders verhielt es sich in diesem Zusammenhang bei dynamischen bzw. sich ständig ändernden Objekten wie zum Beispiel Prozeßobjekte. Es ist nahezu unmöglich, diese MIB-Objekte mit dem realen Systemzustand konsistent zu halten. Dazu hätte bei jedem Prozeßstart ein Objekt instantiiert und dieses bei Prozeßterminierung wieder gelöscht werden müssen. Mit Zugriff auf die Prozeßobjekte über eine Metaklasse „**`MetaProcess`**“, war dieses Problem gelöst. Sobald nun ein Zugriff auf die Metaklasse durchgeführt wird, wird der Bestand an Prozeßobjekten aktualisiert, das anfragende Objekt (in diesem Fall z.B. der Manager) erhält also immer Zugriff auf den aktuellen Systemzustand. *Abbildung 16* zeigt graphisch nochmals diesen implementierungsbedingten „Eingriff“ in das Objektmodell.

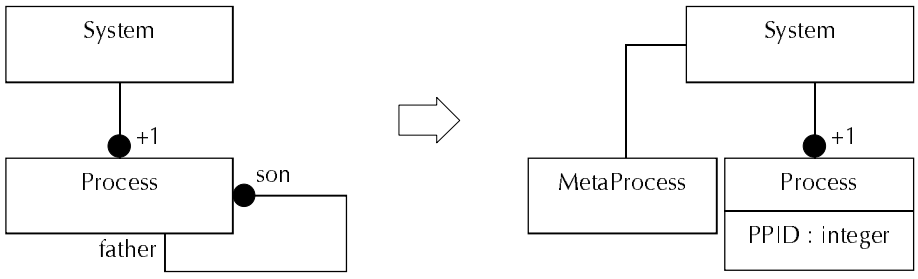


Abbildung 16: Einführung einer Metaklasse zur Verwaltung von Prozeßobjekten

Aus rein objektorientierter Sicht bedeutete dieser Eingriff einen kleinen Rückschritt, zumal die Rollen „father“ und „son“ jetzt nur noch indirekt (über die Variable PPID) gewonnen werden konnten. Aus Implementierungssicht brachte diese Modellierung allerdings eine sehr große Vereinfachung mit sich.

4.5.2 Nachbearbeitung der von StP gelieferten IDL-Schnittstellenbeschreibungen

Leider konnten die von StP gelieferten IDL-Schnittstellenbeschreibungen nicht direkt als Eingabe für den SOM-Compiler verwendet werden. An einigen Stellen war Nachbearbeitung erforderlich. Diese Überarbeitung umfaßte folgende Schritte:

- **Definition spezieller Attributtypen**
  Datentypen, die nicht zu den IDL-Grundtypen gehören, müssen, sofern sie nicht in der Attributdeklaration bestimmt sind, gesondert definiert werden. Im unten angegebenen Beispiel wurden die Typendefinitionen in die Datei **`Types.idl`** ausgelagert. Weiter zu beachten ist, daß der SOM-Compiler zwar Sequenzen kennt, oft jedoch nicht den Datentyp, über dem die Sequenz existiert. So mußte dem SOM-Compiler (z.B. durch „includieren" einer entsprechenden IDL-Datei) bei Auftreten von **`sequence<Processor>`** die Definition von **`Processor`** mitgeteilt werden.

- **Ableitung aller Schnittstellen (interfaces) aus SOMObject**
  Dies ist eine Konvention des SOM-IDL-Compilers. Dieser fordert, daß alle in den IDL-Dateien auftretenden Schnittstellen von SOMObject abgeleitet sind.

- **Ergänzung der IDL-Dateien um einen „implementation"-Abschnitt**
  In diesem Abschnitt muß dem Compiler mitgeteilt werden, in welcher DLL-Datei die spätere Ausgabe stehen soll. Weiter sollte für jedes Nur-Lese-Attribut eine **`noget`**-Anweisung bzw. für jedes Schreib-Lese-Attribut eine **`noget`**- und eine **`noset`**-Anweisung hinzugefügt werden. Dies veranlaßt den Compiler, leere Rahmen für **`get`**- und **`set`**-Operationen zu erzeugen. Bei Fehlen dieser Anweisungen generiert der Compiler automatisch eine interne **`get`**- bzw. eine interne **`get`**- und eine **`set`**-Operation, die den entsprechenden Attributwert liefert bzw. setzt. Dies ist im vorliegenden Fall jedoch meist nicht gewünscht, da es im Fall der „internen" Operationen nicht möglich ist, eigenen Code für die Operationen einzufügen.

Folgender Code zeigt die IDL-Schnittstellenbeschreibung der Klasse **`System`** nach der Überarbeitung gemäß obigen drei Regeln:

```
// StP -- created on Mon Apr 29 15:06:21 1996 for ug201db@sun6 from system oo-mib2_02

#include <somobj.idl>
#include <Types.idl>

#ifndef _System_idl_
#define _System_idl_

// stp class declarations
interface System;
// stp class declarations end


// stp class definition 108
interface System : SOMObject
{
// stp class members
[...]
// stp class members end

#ifdef __SOMIDL__
implementation {
    dllname = "System.dll";

    ClockTicks: noget;
    Contact: noget, noset;
    Date: noget, noset;
    Hardware: noget;
    Location: noget;
    Name: noget;
    Os: noget;
    TimeZone:noget, noset;
    Uptime: noget;
    maxProcessNumber: noget;
```

```
    maxProcessSize: noget;
    curProcessNumber: noget;
    curMaxProcessSize: noget;
    curMaxProcessTime: noget;
     [...]
};
#endif /*__SOMIDL__*/


};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Nachdem nun alle IDL-Dateien derart nachbearbeitet wurden, konnte der SOM-Compiler eingesetzt werden. Dieser erzeugte aus einer IDL-Datei folgende Ausgaben

- **eine .c-Datei**
  diese Datei enthält die Methodenrümpfe, die mit Implementierungscode aufgefüllt werden können
- **eine .ih-Datei**
  dies ist die „Header"-Datei für die zugehörige .c-Datei
- **eine .h-Datei**
  diese Datei ist vom späteren Client zu includieren

Nachfolgend wird ein Ausschnitt einer vom SOM-Compiler erzeugten .c-Datei angegeben, der das Attribut **`ClockTicks`** der Klasse **`System`** betrifft. Zunächst nochmals der betreffende IDL-Code:

```
// StP -- created on Mon Apr 29 15:06:21 1996 for ug201db@sun6 from system oo-mib2_02

#include <somobj.idl>
#include <Types.idl>

#ifndef _System_idl_
#define _System_idl_

// stp class declarations
interface System;
// stp class declarations end


// stp class definition 108
interface System : SOMObject
{
// stp class members
[...]
// stp class members end

#ifdef __SOMIDL__
implementation {
    dllname = "System.dll";

    ClockTicks: noget;
    [...]
};
#endif /*__SOMIDL__*/


};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Daraus generierte der SOM-Compiler folgende .c-Datei:

```
/*
 *  This file was generated by the SOM Compiler and Emitter Framework.
 *  Generated using:
 *      SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_system_Source
#define SOM_Module_system_Source
#endif
#define System_Class_Source

#include "SYSTEM.ih"


/*
 *Method from the IDL attribute statement:
 *"readonly attribute long ClockTicks"
 */

SOM_Scope long  SOMLINK _get_ClockTicks(System somSelf,  Environment *ev)
{
    SystemData *somThis = SystemGetData(somSelf);
    SystemMethodDebug("System","_get_ClockTicks");

    /* Return statement to be customized: */
    return
}
[...]
```

Anstelle des Kommentars **`/* Return statement to be customized */`** konnte nun der entsprechende eigene Code eingesetzt werden.

```
/*
 *  This file was generated by the SOM Compiler and Emitter Framework.
 *  Generated using:
 *      SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_system_Source
#define SOM_Module_system_Source
#endif
#define System_Class_Source

#include "SYSTEM.ih"


/*
 *Method from the IDL attribute statement:
 *"readonly attribute long ClockTicks"
 */

SOM_Scope long  SOMLINK _get_ClockTicks(System somSelf,  Environment *ev)
{
    SystemData *somThis = SystemGetData(somSelf);
    SystemMethodDebug("System","_get_ClockTicks");

    static long tick;
    tick = (long) sysconf( _SC_CLK_TCK );

    return &tick;
}
[...]
```

Diese Datei kann dann (zusammen mit der .ih-Datei) übersetzt werden. Als Ergebnis erhält man dann ein Serverobjekt.

Zur Einordnung in den Gesamtkontext sei an dieser Stelle nochmals auf *Abbildung 7* verwiesen. Die dort angegebenen ersten vier Schritte wurden wie folgt durchgeführt:

1. **Deklaration der Objektklassen mit Hilfe der IDL**
   Die entsprechenden IDL-Schnittstellenbeschreibungen wurden mit StP auf Basis des erstellten Objektmodells erzeugt.

2. **Vorcompilieren der IDL-Datei**
   Dieser Schritt wurde mit Hilfe des SOM-Compilers durchgeführt, der aus der IDL-Datei die entsprechenden Methodenrümpfe generierte.

3. **Hinzufügen von Implementierungscode**
   Dazu mußte nur eine entsprechende Kommentarzeile durch Implementierungscode ersetzt werden

4. **Übersetzen des Codes**
   Damit konnte ein Serverobjekt generiert werden.

Die Schritte 5 bis 7 müssen dann im konkreten Anwendungsfall, d.h. beispielsweise im Einsatz als Systemagent, durchgeführt werden.

5 Arbeiten mit dem StP/OMT-Softwarepaket

5.1 Die Funktionalität und Aufbau des StP/OMT-Softwarepaketes

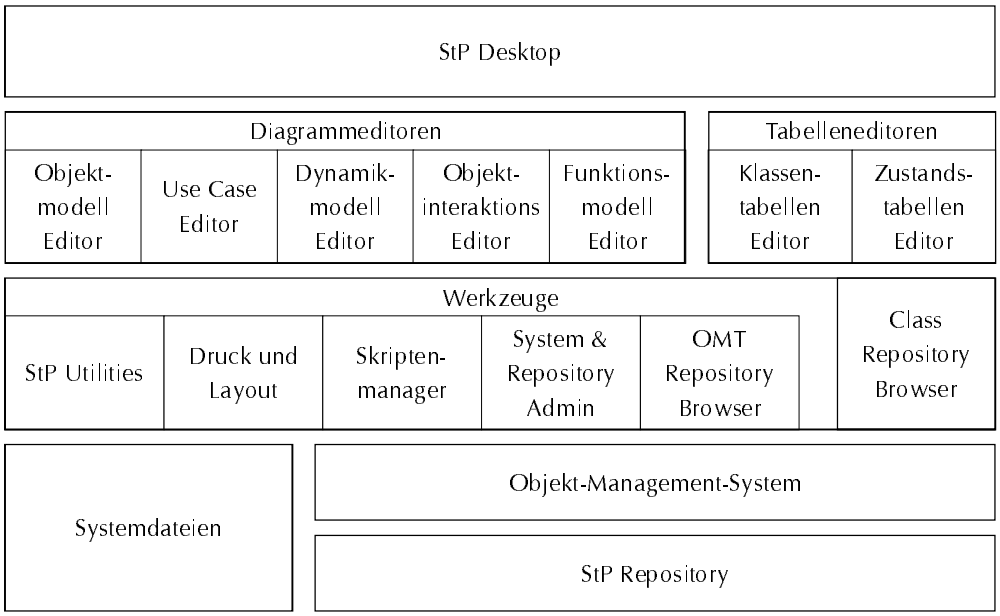


Abbildung 17: Der Aufbau des StP/OMT-Paketes

Das StP-System besteht aus folgenden Komponenten:

StP Desktop

Der StP Desktop ist die Zentrale im StP-Paket. Von ihm aus können alle Editoren und Werkzeuge aufgerufen werden. Genaueres hierzu siehe *Kapitel 5.2: „Der StP Desktop" (Seite 33).*

Class Repository Browser

Mit diesem Browser kann das StP Repository betrachtet und nach OMT Konstrukten durchsucht werden. Genaueres zum Thema „StP Repository" siehe *Kapitel 5.4.2 (Seite 36).*

Diagrammeditoren

Die Editoren erlauben, die verschiedenen Diagramme im OMT-Modell zu bearbeiten. Dazu gehören Editoren für das Objektmodell, das Dynamikmodell, das Funktionsmodell, ein Editor für Use Cases sowie ein Editor zur Beschreibung von Objektinteraktionen. Das Kapitel 5.3 („Die Editoren") geht genauer auf diese Komponenten ein.

Tabelleneditoren

Analog zu den Diagrammeditoren erlauben es die Tabelleneditoren, Tabellen des OMT-Modells zu bearbeiten. Dazu gehören der Klassentabelleneditor und der Zustandstabelleneditor.

Werkzeuge

Die Werkzeuge unterstützen den Benutzer darin, die erstellten Diagramme und Tabellen weiterzuverarbeiten bzw. auszugeben. Dazu gehören unter anderem ein StP Repository Browser (*siehe Kap. 5.4.2*), ein Skriptenmanager zur Erzeugung von Quellcode und Dokumentationen usw.

Systemdateien

Jede Systemdatei enthält Informationen zu jeweils einem Diagramm bzw. einer Tabelle. Diagrammdateien speichern Daten zu den Symbolen eines Diagramms, Tabelleneditoren speichern Zellinhalte. Mehrfach vorkommende Elemente werden auch mehrfach abgespeichert.

Object Management System

Mit dem Object Management System kann der Benutzer seine OMT-Modelle bequem verwalten. Mit einem Browser ist es möglich, Abfragen zu erstellen, die entweder auf StP/OMT Konstrukten (in Form einer *Query-by-example*) basieren oder mit einer eigenen Abfragesprache, der *Object Management System query language*, formuliert werden.

StP Repository

Das StP Repository ist der zentrale Datenspeicher im StP-System. Hier werden alle Daten bezüglich Diagrammen und Tabellen gehalten. Mehrfach in Diagrammen oder Tabellen vorkommende Objekte werden hier nur einfach gespeichert. Genaueres zum StP Repository siehe *Kapitel 5.4.2*.

5.2 Der StP Desktop

Nach Aufruf von Software through Pictures mittels des Kommandos

```
stp [&]
```

erscheint der StP Desktop auf dem Bildschirm. Er ist die Zentrale im StP-Paket. Von ihm aus können alle Editoren und Werkzeuge *(siehe Abbildung 17)* aufgerufen werden.

Abbildung 18 zeigt das StP-Desktop-Fenster:

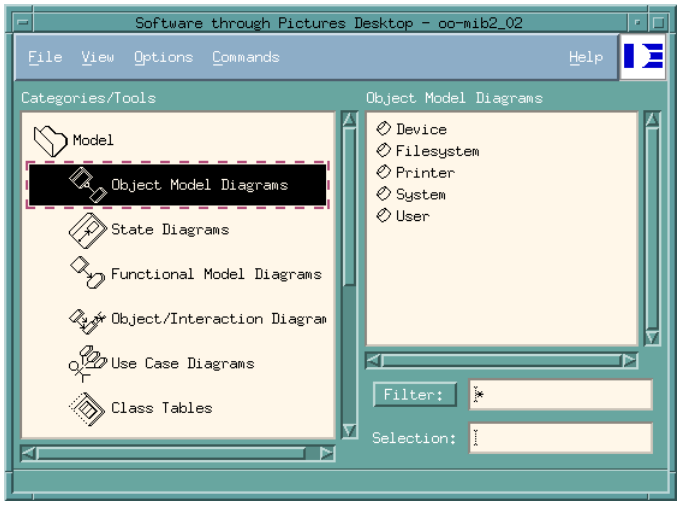


Abbildung 18: Der Software through Pictures Desktop

Der StP Desktop enthält in dem Feld „Categories/Tools“ Icons, die die einzelnen Module des StP-Paketes repräsentieren. Um nun ein **neues Projekt** zu starten, muß zunächst ein neues „System“ bestimmt werden. Dies geschieht folgendermaßen: Man wählt im StP Desktop die Kategorie „Administer“ [7]. Anschließend selektiert man „Administer System“. Aus dem „Commands“-Menü wählt man den Menüpunkt „Create System“, woraufhin sich eine Dialogmaske öffnet, in der man das neue Projekt anlegen kann.

[7] Diese ist in *Abbildung 18* nicht zu sehen; dazu müßte in dem Feld „Categories/Tools“ nach unten gescrollt werden.

Bevor Sie nun mit der Arbeit an einem neuen oder bereits bestehenden Projekt beginnen können, müssen Sie dieses Projekt als „aktiv“ bestimmen. Wählen Sie hierzu im Menü „File“ den Menüpunkt „Set Project/System“

Bei Anklicken eines der Icons, erscheinen im Feld daneben die bereits vorhandenen Modelle oder Tabellen. Einen Editor öffnet man nun, indem man eines der bestehenden Modelle selektiert und anschließend im Menü „Commands“ den Menüpunkt „Edit Diagram...“ auswählt bzw. „Start New Editor...“ um ein neues Objekt der gewählten Kategorie zu erstellen.

Die **Codeerzeugung** kann ebenfalls vom StP Desktop aus eingeleitet werden. Dazu muß im Feld „Object Model Diagrams“ ein Objektmodell ausgewählt werden. Anschließend kann durch Aufruf eines entsprechenden Menüpunktes im Menü „Commands“ eine Ausgabe der gewünschten Art erzeugt werden. In der Version 3 von StP können Codes für folgende Sprachen generiert werden:

- C++
- Smalltalk
- IDL
- Ada-83

5.3 Die Editoren

Die Diagrammeditoren haben alle ein ähnliches äußeres Erscheinungsbild. In *Abbildung 19* ist exemplarisch das Fenster des Objektmodelleditors zu sehen. Aus der Elementleiste können, je nach Diagrammeditor, entsprechende Diagrammelemente (z.B. Klassensymbole, Beziehungspfeile etc.) in die Arbeitsfläche gezogen[8] werden. Mit Hilfe der Bildlaufleisten kann der jeweils gewünschte Ausschnitt aus dem Modell angezeigt werden.

Ein Objekt wird zumeist dadurch bezeichnet, daß das betreffende Objekt mit dem Mauszeiger selektiert und anschließend der Bezeichner über Tastatur eingegeben wird.

Objektmodelleditor

Mit dem Objektmodelleditor kann, wie die Bezeichnung schon sagt, das Objektmodell des Projektes erstellt werden. Es können Klassen definiert werden, deren Attribute eingefügt, Klassenoperationen festgelegt sowie Beziehungen (Assoziationen, Aggregationen, Vererbungen) zwischen Klassen modelliert werden.

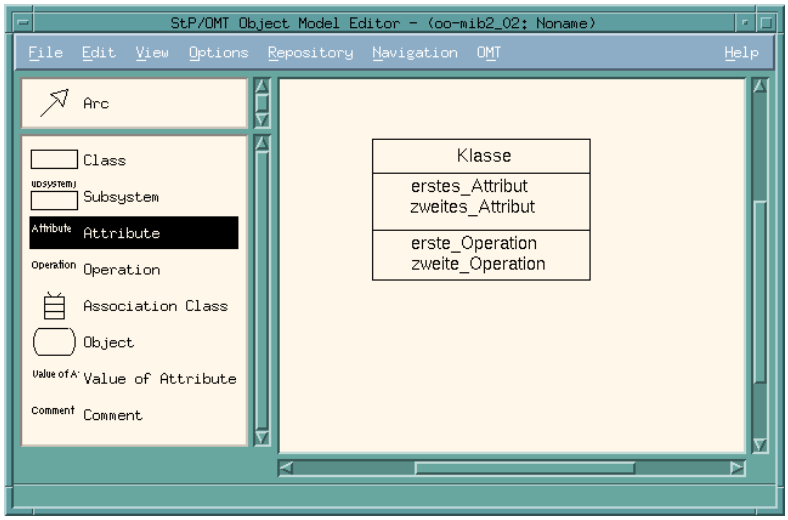


Abbildung 19: Aufbau eines StP-Editorfensters (am Beispiel des Objektmodelleditors)

[8] „gezogen“ soll heißen: das entsprechende Objekt wird mit gedrückter Maustaste in den Arbeitsbereich geführt.

Use-Case-Editor

Zur Modellierung von Use Cases, Ereignisfluß- und Ereignisverfolgungsdiagrammen kann dieser Editor eingesetzt werden. Use Cases können in Ereignisszenarien zerlegt werden. Darüber hinaus können für Ereignisse Echtzeitvorgaben getroffen werden.

Dynamikmodelleditor

Mit diesem Editor ist es möglich, für die wichtigen Klassen eines Modells jeweils ein Zustandsdiagramm anzulegen, welches das dynamische Verhalten der betreffenden Klasse im Objektmodell über die Zeit beschreibt.

Objektinteraktions-Editor

Dieser Editor ermöglicht es dem Benutzer eine spezielle Art von Objektdiagrammen zu erstellen, die die Interaktion der einzelnen Objekte beschreiben. Darüber hinaus können Interaktionsdiagramme erstellt werden, die Objektinteraktionen graphisch anders aufzeigen. Die eine Diagrammart kann auf Wunsch automatisch aus der anderen erzeugt werden. Objekt- und Interaktionsdiagramm werden automatisch synchronisiert.

Funktionsmodelleditor

Das Funktionsmodell kann mit diesem Editor in DeMarco/Yourdon und Rumbaugh-Notation erstellt werden. Prozesse, Datenspeicher und Objekte können ebenso modelliert werden wie Daten-, Kontroll- und Objektflüsse.

Klassentabelleneditor

Mit dem Klasseneditor können die Attribute und Operationen für eine einzelne Klasse definiert werden. Darüber hinaus ist es möglich, für bestimmte Ausgaben spezifische Einstellungen vorzunehmen. Dazu gehören unter anderem Einstellungen für C++ (z.B. ob ein Attribut **`privilege`** oder **`const`** ist), Smalltalk, Ada-83 usw.

Zustandstabelleneditor

Der Zustandstabelleneditor verwendet die gleichen Elemente wie der Dynamikmodelleditor (s.o.). In einer tabellarischen Übersicht können hier ebenfalls Zustände, deren Änderungen, Eingangs- und Ausgangsereignisse modelliert werden.

5.4 Die Informationsspeicherung in StP

5.4.1 Die Systemdateien

StP legt für jedes erstellte Diagramm, jede Tabelle und Anmerkung, die der Benutzer seinem OMT-Modell hinzufügt, eine eigene Datei an. Diese Diagrammdateien enthalten Informationen über die Objektbeziehungen, Bezeichnungen der Beziehungen, Struktur der Diagramme usw. Tabellendateien enthalten Informationen über die Inhalte der Tabellenzellen. In Anmerkungsdateien werden – wie der Name schon sagt – die Anmerkungen gespeichert, die der Benutzer zu Diagrammen oder Tabellen angibt.

In den Systemdateien wird jede Komponente eines Diagramms oder einer Tabelle für sich gespeichert. Das heißt also: tritt zum Beispiel eine Klasse in einem Diagramm mehrfach auf, so wird sie in der Diagrammdatei auch mehrfach erfaßt. Dabei wird allerdings die Information, die jeweils gleich bleibt, nicht mehrfach gespeichert. Die Erfassung erfolgt durch Einfügen mehrerer Referenzen auf die entsprechende Klasse im Repository (s.u.).

5.4.2 Das Repository

Das Repository ist der zentrale Datenspeicher im StP-System. Hier werden die Informationen über die einzelnen Komponenten des OMT-Modells gehalten. Dabei ist jede Komponente jeweils nur einmal gespeichert, egal wie oft sie in den Diagrammen, Tabellen usw. erscheint. Diagramm- oder Tabellendateien (s.o.) greifen auf diese Komponenten über Referenzen zu. Diese Vorgehensweise garantiert, daß die Inhalte aller Systemdateien konsistent sind. Eine bestimmte Klasse, die z.B. in mehreren Diagrammen – unter Umständen dort wiederum mehrfach – auftritt, hat stets die gleiche Struktur. Widersprüchliche Eingaben zu einer Klasse sind also unmöglich.

Um eine OMT-Komponente in das Repository aufzunehmen, genügt es, das Diagramm, die Tabelle oder die Anmerkung abzuspeichern, in der die Komponente zuletzt bearbeitet wurde.

5.5 Der Entwurf eines OMT-Modells

In den meisten Fällen wird unter StP ein OMT-Modell folgendermaßen entwickelt: Zunächst wird das Objektmodell erzeugt. Dabei finden vor allem der Objektmodell- und der Klassentabelleneditor Verwendung. Unter Umständen setzt man, sofern man bei der Objektmodell-Entwicklung zum Auffinden von Klassen, Attributen und Operationen mit Use Cases arbeitet, in dieser Phase auch den Use Case Editor ein. Mit Hilfe dieser drei Werkzeuge können nun alle Klassen, die Operationen und Attribute sowie die statischen Beziehungen untereinander modelliert werden.

Zentraler Editor ist in vielen Fällen der Objektmodelleditor. Von ihm aus werden dann die anderen Editoren aufgerufen. Dynamikmodelleditor, Funktionsmodelleditor und die Editoren zur Erzeugung von Ereignisdiagrammen helfen dann, das OMT-Modell zu verfeinern und weitere Vorgaben für die Codeerzeugung zu machen. *Abbildung 20* zeigt nochmals graphisch die Zusammenhänge der einzelnen OMT-Modelle im Rahmen der Softwaremodellierung mit dem StP-Werkzeug.

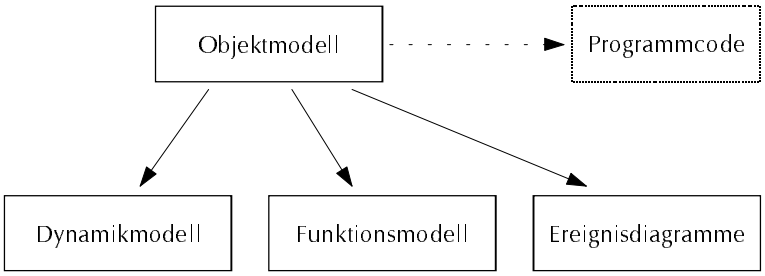


Abbildung 20: Zusammenhänge zwischen den einzelnen OMT-Modellen

Wird von einem Editor in einen anderen gewechselt, so erscheint meist die gerade bearbeitete Komponente in einer anderen Darstellungsform. Wird zum Beispiel vom Objektmodelleditor in den Klassentabelleneditor gewechselt, so erscheint das gerade bearbeitete Objekt in Tabellenform.

Was das Hin- und Herschalten zwischen den Editoren betrifft, so ist zu beachten, daß in StP nicht von jedem beliebigen Editor X in einen beliebigen anderen Editor Y gewechselt werden kann. Die Tabelle in Abbildung 21 gibt darüber Aufschluß, von welchem Editor jeweils in welchen anderen navigiert werden kann (eine Markierung bedeutet, daß von dem Editor links in den Editor oben gewechselt werden kann). Bemerkenswert ist hierbei zum Beispiel, daß es nicht möglich ist, vom Objektmodelleditor in den Use Case Editor zu schalten. Man sollte die Use Case-Modellierung also vollständig abschließen, ehe man mit der Objektmodellierung beginnt.

Wohl ist es aber zum Beispiel möglich, im Objektmodelleditor eine Klasse auszuwählen, anschließend den Klassentabelleneditor aufzurufen, um die markierte Klasse genauer zu modellie-

ren, diese Änderungen abzuspeichern und danach wieder in den Objektmodelleditor umzuschalten, um dort wiederum die Anzeige gemäß den eben gemachten Änderungen aktualisieren zu lassen.

		nach							
		OME	UCE	DME	OIE	FME	CTE	STE	PE
von	Objektmodelleditor (OME)	●		●	●	●	●		●
	Use Case Editor (UCE)	●	●	●			●		●
	Dynamikmodelleditor (DME)	●	●	●			●	●	●
	Objektinteraktionseditor (OIE)	●	●	●	●		●		●
	Funktionsmodelleditor (FME)	●				●	●		●
	Klassentabelleneditor (CTE)	●		●		●	●		●
	Zustandstabelleneditor (STE)			●				●	
	Programmierumgebung (PE)								●

Abbildung 21: Tabellarische Auflistung der Navigationsmöglichkeiten durch die verschiedenen StP-Editoren

5.6 Beurteilung

Software through Pictures ist ein mächtiges Werkzeug für die computerunterstützte Softwarekonstruktion. Für alle Phasen der Softwareentwicklung stellt es leistungsfähige Editoren bereit. Diese unterstützen komfortabel die Entwicklung und Wartung des Datenbestandes eines Softwareprojektes. Die Erstellung graphischer Modelle und die „Nachbearbeitung“ in Tabellen werden ebenso unterstützt wie die Dokumentation des Projektes.

Nicht unerwähnt bleiben soll natürlich die leistungsfähige Codeerzeugung. Mit C++, Smalltalk und Ada-83 werden drei wichtige objektorientierte Programmiersprachen berücksichtigt. Die Tatsache, daß seit der Version 3 von StP auch CORBA-IDL-Schnittstellenbeschreibung erzeugt werden kann, machte dieses Werkzeug für dieses Praktikum geeignet. Es liegt an der eingeschränkten Syntax von IDL, daß Beziehungen im Objektmodell nur mangelhaft in der ausgegebenen Schnittstellenbeschreibung wiedergegeben werden konnten. Nichtsdestotrotz brachte der Einsatz von StP eine gewaltige Arbeitserleichterung mit sich. Änderungen im Objektmodell konnten schnell und unproblematisch vorgenommen werden, hinsichtlich der Syntax und Semantik überprüft und entsprechend als IDL-Schnittstellenbeschreibung ausgegeben werden. Wie in *Kapitel 4.5.2* beschrieben, muß dieser Code dann allerdings gemäß dem verwendeten IDL-Compiler ein wenig nachbearbeitet werden. Bei häufigerem Einsatz (beispielsweise im Rahmen der Weiterentwicklung des Objektmodells) wäre es zu überlegen, ob nicht die Möglichkeit in Anspruch genommen werden sollte, das Ausgabemodul von StP derart umzukonfigurieren, daß eine IDL-Ausgabe erzeugt wird, die der Eingabe des IDL-Compilers entspricht.

Einziger Wermutstropfen an StP ist die manchmal etwas zu lange Antwortzeit. Als Folge des Umfanges dieses Werkzeuges (für jeden Entwicklungsschritt steht ein eigener Editor zu Verfügung, welcher zur Modellierung erst gestartet wird) muß zwangsläufig die Performanz etwas leiden. Bei einer Softwareentwicklung, die vollständig mit diesem Werkzeug durchgeführt wird, ist dies durchaus tolerierbar. Beschränkt man sich jedoch – wie in diesem Praktikum – im wesentlichen auf die Erstellung eines Objektmodells, so muß man StP deshalb fast als überdimensioniert bezeichnen.

6 Ausblick

Die objektorientierte Modellierung der Management Information Base bringt einige Vorteile gegenüber der bisherigen ASN.1-Notation mit sich. Die Semantik der MIB-Komponenten wird besser dargestellt. Es gibt keine mißverständlichen „Pushbutton"-Variablen mehr – an deren Stelle sind nun Operationen getreten. Eine Aktion kann nun dadurch ausgelöst werden, daß eine Operation aufgerufen wird, nicht mehr dadurch, daß an eine Variable ein bestimmter Wert zugewiesen wird. Es ist nun ebenfalls besser möglich, Beziehungen zwischen Managementobjekten wiederzugeben und dadurch die Managementinformation zu strukturieren. So werden Platten und Partitionen beispielsweise nicht mehr unabhängig voneinander gespeichert. Vielmehr wird nun die Beziehung „Disk enthält Partitionen" so in dem Informationsmodell gespeichert. Ein weiterer Vorteil des objektorientierten Ansatzes ist die Wiederverwendbarkeit von Code durch Vererbung. Es ist leicht möglich, spezialisierte Komponenten von bestehenden in das Modell aufzunehmen, wobei nur die „Besonderheit" neu implementiert werden muß; der Rest kann vererbt werden.

Die relativ leichte Erweiterung der MIB ist in jedem Fall ein gewichtiger Vorteil des in diesem Praktikum verfolgten Ansatzes. Eine Änderung kann mit dem StP/OMT-Werkzeug schnell durchgeführt werden. Sie läuft in folgenden fünf Schritten ab:

1. Ergänzung des Objektmodells
2. Erzeugen einer IDL-Schnittstellenbeschreibung
3. Nachbearbeitung der IDL-Ausgabe
4. Hinzufügen von Implementierungscode
5. Compilieren

Die IDL-Ausgabe von StP kann nach geringfügiger Nachbearbeitung als Eingabe für einen IDL-Compiler verwendet werden. Wie schon in *Abschnitt 5.6* erwähnt, wäre es überlegenswert, StP derart zu konfigurieren, daß eine IDL-Schnittstellenbeschreibung erzeugt wird, die direkt als Eingabe für den gewünschten IDL-Compiler dienen kann.

Ein Ansatz zur Erweiterung des Objektmodells zeigt sich etwa im Zusammenhang mit der Klasse **`Generic_Device`**. Hier gibt es noch unzählige Geräte die von **`Generic_Device`** abgeleitet werden könnten. Dazu gehören zum Beispiel Komponenten wie Schnittstellen, Controller, Grafikkarten, Soundkarten etc.

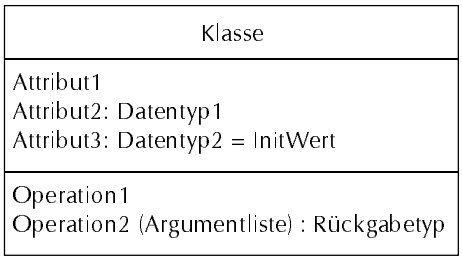
Ebenfalls ist denkbar eine Erweiterung der Beziehungsstruktur der Klassen. Für Multiprozessorsysteme, bei denen jedem Prozessor ein bestimmter Speicher zugeteilt ist, könnte zum Beispiel eine Beziehung zwischen der Klasse **`Storage`** und der Klasse **`Processor`** eingefügt werden.

Anhang A: Abkürzungen

ASN.1	Abstract Syntax Notation One
BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
DSOM	Distributed System Object Model
IDL	Interface Definition Language
MIB	Management Information Base
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
ORB	Object Request Broker
RFI	request for information
RFP	request for proposal
RPC	Remote Procedure Call
STP	Software through Pictures
SOM	System Object Model

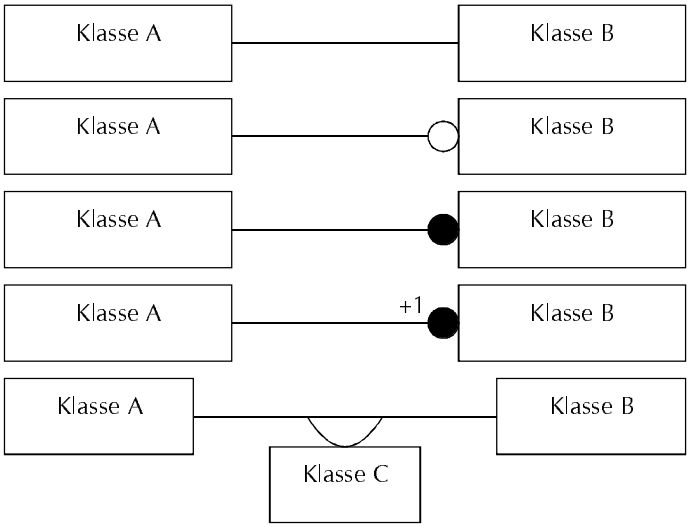
Anhang B: Teil der OMT-Notation nach Rumbaugh

Klassendefinition, Attribute und Operationen



Eine Klasse wird dargestellt durch ein dreigeteiltes Rechteck. Das obere Feld enthält den Klassenbezeichner, das mittlere Feld die Attribute, das untere Feld die Operationen.

Assoziationen (Beziehungen)



1:1 – Beziehung

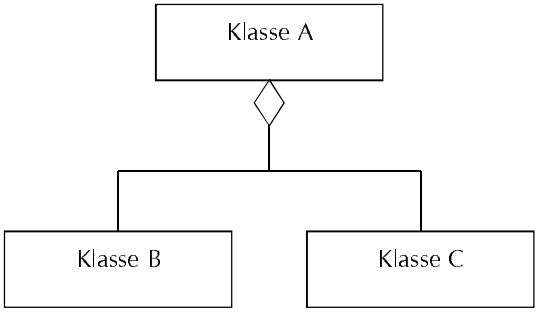
1:n – Beziehung (n=0 oder n=1)

1:n – Beziehung (n≥0)

1:n – Beziehung (n≥1)

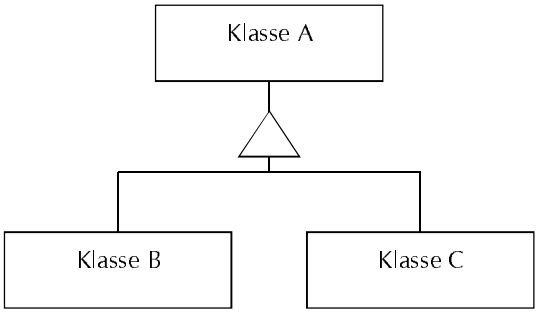
Klasse C ist Assoziationsklasse der Assoziation zwischen Klasse A und Klasse B

Aggregation (Enthaltensein)



Klasse A enthält die Klassen B und C

Generalisierung (Vererbung)



Die Klassen B und C sind von Klasse A abgeleitet, d.h. sie erben alle Attribute und Operationen von Klasse A.

Die Klassen B und C sind Spezialfälle von Klasse A

Klasse A ist Verallgemeinerung von Klasse B und C

Anhang C: Literatur

Im folgenden wird hier die zu den einzelnen Themen verwendete Literatur angegeben:

CORBA

[DB1] Dennis Byron, „Object Management Group Common Object Request Broker Architecture“, McGraw-Hill, Januar 1994

[DB2] Dennis Byron, „Object Management Architecture“, McGraw-Hill, Januar 1994

[MB1] Mark Betz, „Interoparable Objects“, D. Dobb's Journal, Oktober 1994

[MST1] Michael Stal, „Der Zug rollt weiter: CORBA 2.0 und weitere OMG-Standards“, iX, Mai 1995

[OMG1] Object Management Group, „Object Management Architecture Guide“, Second Edition, September 1992

[TB1] Thorsten Beyer, „Objektbörse: CORBA - OMG-Standard für verteilte Objekte“, iX, Februar 1993

SOM

[RODH] Robert Orfali, Dan Harkey, „OS/2 Client/Server Survival Guide“, Van Nostrand Reinhold

Software through Pictures

[IDE1] Interactive Development Environments, „StP Technical White Paper“

[IDE2] Interactive Development Environments, „Software through Pictures: Information Modeling“

[IDE3] Interactive Development Environments, „Software through Pictures: Object Modeling Technique“

[IDE4] Interactive Development Environments, „Software through Pictures: Test Case Generator“

Objektmodellierung

[EB1] Ernst Bötsch, „Objektorientierte Modellierung von UNIX-Rechnern unter dem Gesichtspunkt des System-Managements“, Projektbeschreibung MNM-Team, Oktober 1995

[JR1] James Rumbaugh, „Using Use Cases to Capture Requirements“, Journal of Object-Oriented Programming, Juli 1994

Anhang D: Abbildungsverzeichnis

Abbildung 1: Vorgehensweise zur Gewinnung eines CORBA-konformen Systemagenten aus der ASN.1-Schnittstellenbeschreibung und dem bestehenden SNMP-Agentencode 4
Abbildung 2: Die OMG Object Management Architecture 5
Abbildung 3: Die Architektur eines Object Request Brokers 7
Abbildung 4: Funktionsprinzip des Basic Object Adapter 8
Abbildung 5: Einordnung der CORBAfacilities in die OMA 10
Abbildung 6: SOM Klassen, Metaklassen und Objekte 12
Abbildung 7: Entwicklungsschritte für CORBA Dienste – von der IDL-Beschreibung zu den Interface-Stubs 13
Abbildung 8: Ausschnitt aus der Use Case Modell 15
Abbildung 9: Die Klasse „System“ 18
Abbildung 10: Die Klasse „Storage“ mit der ihr übergeordneten Systemklasse 19
Abbildung 11: Die beiden Prozessorklassen „Cpu“ und „CpuGeneral“ mit der ihnen übergeordneten Systemklasse 20
Abbildung 12: Ausschnitt aus der ersten Objekt-MIB 21
Abbildung 13: Die Beziehungsstruktur in der optimierten Objekt-MIB 24
Abbildung 14: „Quota“ als Assoziationsklasse der Beziehung User-Filesystem 24
Abbildung 15: Ausschnitt aus der optimierten Objekt-MIB 25
Abbildung 16: Einführung einer Metaklasse zur Verwaltung von Prozeßobjekten 27
Abbildung 17: Der Aufbau des StP/OMT-Paketes 32
Abbildung 18: Der Software through Pictures Desktop 33
Abbildung 19: Aufbau eines StP-Editorfensters (am Beispiel des Objektmodelleditors) 34
Abbildung 20: Zusammenhänge zwischen den einzelnen OMT-Modellen 36
Abbildung 21: Tabellarische Auflistung der Navigationsmöglichkeiten durch die verschiedenen StP-Editoren 37

Anhang E: Die Objektmodelle

Die folgenden zwei Seiten wurden mit dem CASE-Tool „Software through Pictures“ erstellt. Sie zeigen eine vollständige Übersicht über die beiden im Rahmen dieses Praktikums erstellten Objektmodelle.

Die erste Seite zeigt die „naive“ Objekt-MIB, die mechanisch aus der SNMP-MIB modelliert wurde. Für genauere Informationen zu diesem Objektmodell sei an dieser Stelle auf *Kapitel 4.2* verwiesen.

Die zweite Seite enhält eine graphische Darstellung der optimierten Objekt-MIB. Genaueres zur Optimierung ist in *Kapitel 4.4* zu finden.